TECHNISCHE UNIVERSITÄT
CHEMNITZ

FPGA-based Speed Limit Sign Detection

Master Thesis

Submitted in Fulfillment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Reham Tallawi
Student ID: 386157
Date: 03.07.2017

Supervising tutor: Prof. Dr. W. Hardt
Supervising tutor: Dipl. Ing. Stephan Blokzyl

Acknowledgment

I would like to express my sincere gratitude to my advisor and mentor Dpl.-Ing. Stephan Blokzyl, thank you for all your support, advice, guidance, and above all your patience you provided me during my work with you as a HIWI student, and in the masters internship, and finally in the masters thesis. I really appreciate all what you have done and for that I am eternally grateful. I also would like to thank Prof. Dr. Wolfram Hardt Chair of the Computer Engineering department at Technischer Universitt Chemnitz (TUC) for his continuous support and encouragement to his students. Finally, I want to thank my family specially my mom and dad for all their support during this wonderful journey. You have always believed in me, no matter what.

Abstract

This thesis presents a new hardware accelerated approach using image processing and detection algorithms for implementing fast and robust traffic sign detection system with focus on speed limit sign detection. The proposed system targets reconfigurable integrated circuits particularly Field Programmable Gate Array (FPGA) devices. This work propose a fully parallelized and pipelined parallel system architecture to exploit the high performance and flexibility capabilities of FPGA devices.

This thesis is divided into two phases, the first phase, is a software prototype implementation of the proposed system. The software system was designed and developed using C++ and OpenCV library on general purpose CPU. The prototype is used to explore and investigate potential segmentation and detection algorithms that might be feasible to design and implement in hardware accelerated environments. These algorithms includes RGB colour conversion, colour segmentation through thresholding, noise reduction through median filter, morphological operations through erosion and dilation, and sign detection through template matching. The second phase, a hardware-based design of the system was developed using the same algorithms used in the software design. The hardware design is composed of 20 image processing components each designed to xxx fully parallelized and pipelined xxx. The hardware implementation was developed using VHDL as the hardware description language targeting a Xilinix Virtex-6 FPGA XC6VLX240T device. The development environment is Xilinx ISE®Design Suite version 14.3.

A set of 20 640x480 test images was used as the test data for the verification and testing of this work. The images was captured by a smart-phone camera in various weather and lightning conditions. The software implementation delivered speed limit detection results with a success rate of 75%. The hardware implementation was only simulated using Xilinx ISE Simulator (ISim) with a overall system latency of 12964 clock cycles. According to the Place and Route report the maximum operation frequency for the proposed hardware design is 71,2 MHz. The design only utilized 2% of the slice registers, 4% of the slice Look up Tables (LUT), and 11% of the block memory. This thesis project concludes the work based on the provided software and hardware implementation and performance analysis results. Also the conclusions chapter provides recommendations and future work for possible extension of the project.

Keywords: FPGA, traffic sign detection, image processing, VHDL, hardware acceleration

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ADAS Advanced Driver Assistance Systems

AISC Application Specific Integrated Circuits

BGR Blue Green Red

CB Connection Box

CC Cross-Correlation

CHT Circular Hough Transform

CLB Configurable Logic Block

CMYK Cyan, Magenta, Yellow, Black

COE Coefficient File

CPU Central Processing Unit

DRAM Dynamic Random Access Memory

EO Electro Optical

FIFO First In First Out

FPGA Field Programmable Gate Array

FRS Fast radial symmetry

HD High Definition

HDL Hardware Description Language

HSV Hue Saturation Value

HT Hough Transform

IP-core Intellectual Property core

ISE Integrated Synthesis Environment

ISim ISE Simulator

Lab Lightness a b

LUT Look-up Table

NCC Normalized Cross-Correlation

RAM Random Access Memory

RGB Red Green Blue

ROI Region of Interest

ROM Read Only Memory

RTL Rigiter Transfer Level

SAD Sum of Absolute Differences

SB Switch Box

SoC System on Chip

SRAM Static Random Access Memory

TSR Traffic Sign Recognition

VHD VHDL Design File

VHDL Very High Speed Integrated Circuit HDL

VLSI Very Large Scale Integration

1 Introduction

Over the past few decades there have been an increasing interest to develop and enhance traffic recognition systems in vehicles. Considering the significant increases in the number of street vehicles and automobiles in large and metropolitan cities across the world, the need for a reliable and safe systems becomes crucial. The systems not only should deliver safe and secure operations, but it should be intuitive and intelligent enough to provide road surveillance, allow sign recognition, and provide drivers with guidance through out the journey. One of the main reasons for this shift in the market, is due to the highly advancement in computer hardware, providing affordable technologies with high computational and processing capabilities [2].

One of the most extensively researched subjects in the automotive industry, is the advance driving assistance systems (ADAS) particularly traffic sign recognition systems (TSR) [3]. TSR is an automatic subsystem of ADAS located inside vehicles that supports and guides vehicle drivers during their journey. TSR provides visual aids to check of any traffic warning signs (such as speed limits), guide signs, and regulation signs. TSR can provide drivers with safety and comfort as well as alert drivers for undertaking inappropriate driving decisions such as over speeding, taking incorrect turn, entering one way streets, and passing other vehicles in a no-pass zone. At the moment, there are some high end luxury car models already have TSR systems embedded into their system offering automated detection and recognition of a certain class of traffic signs [4].

The development of TSR systems (usually software based systems) is an exhausting procedure as the system requires to keep track of the changes in traffic signs on the road, not to mention the system shall detect the signs regardless of the weather and lightning conditions [5] [6]. Initially, the system captures streaming images of the road from a high-speed and high-resolution camera/s mounted on the front of a moving vehicle. The captured video streams is then passed by the TSR processing unit for region of interest analysis, detection, recognition, and tracking. Finally, the recognized sign is displayed on the dashboard of the vehicle to alert drivers of incoming signs. In image processing, the traffic sign recognition system is categorized into three main stages image segmentation, detection, classification/recognition, and tracking [3]. In each stage the incoming video streams is processed and passed on to the next stage for further analysis, until the final output is computed and presented to the screen. The complete detection and recognition process requires very high processing power, high speed, and large number of resources in order to deliver the required task in real-time and with high performance. These tough requirements is

due to two factors. First, due to the high complexity operations of computer vision and image processing algorithms used for detection, recognition, and tracking. Second, the requirements are due to the high quality, large sized video streams captured from the mounted camera sensors.

1.1 Motivation

In software-based TSR systems, the processing of high-volume high-resolution streaming videos can be a challenging task particularly when utilizing complex image processing algorithms that require fast real-time performance. From the low-level design point of view software-based systems using general purpose processors as well as micro controllers, have fixed hardware element structure, in the sense that resources such as memories, peripherals and connections are predefined and cannot be changed during system development. Also due to the nature the design architecture of software-based systems, the instructions are executed sequentially. Due to these two issues, the overall performance speed can be slower specially when performing streaming video and image processing for high-critical and real-time applications. Accordingly, the need for an alternative high-performance parallel processing technology becomes critical, hence hardware-based reconfigurable technologies. With the current advancement and development in electronic circuitry, reconfigurable devices gained a lot of popularity in recent years, particularly Field Programmable Gate Arrays (FPGAs) devices. These devices are capable of delivering high performance, power efficient, real-time, and re-programmable complex hardware accelerated computing capabilities [7]. And for that reason, they are widely used in highly critical, timing-constraining applications that needs high processing power and performance. And as such, with the advanced FPGA architectures, the devices can deliver high performance results when targeting the development of TSR systems.

1.2 Goal

The main goal of this work is to introduce fully parallelized and pipelined hardware accelerated traffic sign detection system. To achieve fast high performance, low power, and efficient system, this work is implementing FPGA-based traffic sign detection system. The proposed system should provide:

- Speed limit sign detection: The system shall be able to detect successfully the speed limit signs in images through Template matching algorithm
- High-performance: The hardware design shall be fully parallelized and pipelined
- Light-Invariant capabilities: The system should be able to detect speed limit signs regardless the weather and lightning condition

- Robustness: The system should be able to maintain executing it's operations correctly as well as work with images that has high tendency to noise and distortions.

Also, a software-based system shall be developed before the hardware implementation to verify the correctness and reliability of the algorithms. The algorithms later on will be implemented on hardware. A testing and verification of the results obtained from the software system and hardware implementation should be performed as a secondary task to compare the performance measures and benefits of the FPGA as compared to the software system.

1.3 Thesis structure

The thesis is structured as follows: Chapter two introduces background information about the German traffic system, image processing, and reconfigurable integrated circuits. Chapter three explores the latest state of the art in the field of image processing particularly feature extraction and classification in addition to, the latest in technology in FPGA reconfigurable devices. Chapter four describes the overall system architecture and design of the proposed parallel system. Chapter five presents the software prototype and hardware implementation of the system. Chapter six presents the software and hardware results and analysis. Finally, chapter seven summarizes the thesis project and discuss future work.

2 Fundamentals

2.1 German traffic system

Germany has one of the most comprehensive and standardized traffic system in the world, and which comply to the European Union traffic system standards and regulations. All traffic signs have a simple universal standard shape, size, and colour that makes it very easy for anyone with basic knowledge in traffic system to understand. The German traffic signs are grouped into three main categories regulation signs, guide signs, and warning signs. The figure 1 below shows an example of each category of the German traffic sign system.

Figure 1: German traffic sign illustration [8]

Thanks to the standardised system and the clear distinction between different signs in terms of size, colour, and shape. Together with the advance available technologies, it is easy to design and develop reliable traffic sign recofnition systems.

2.2 Digital image processing

Digital image processing is a crucial methodology to manipulate and enhance digital contents by the usage of computer algorithms. Digital image processing has

the capability to be modeled into different systems depending on image representation(2D, 3D, or more). At first, the performance cost for executing image processing algorithms on images was very high when the discipline first started in the 60s. This was due to the limited available computer resources present at that time. Performance costs started to decrease in the 70's when dedicated hardware, cheaper and faster computer technology became available. Today, with the latest innovations and the latest state of the art in processors, memories and hardware technologies design; performance costs not only reduced significantly but systems became more efficient, fast and robust. Digital image processing algorithms now can execute in real time and in parallel (thanks the advancement in processor's core technologies and reconfigurable hardware devices) hence increasing the speed significantly and output is produced in a timely manner.

Applications of digital image processing methods are very broad and vast as it depends on the designated purpose of application. Some examples of image processing algorithms includes but not limited to: segmentation, classification, convolution, feature detection and extraction, pattern recognition, noise reduction, color space conversion, etc... In the following sections, several popoular image processing methodologies and approaches will be discussed in details.

2.2.1 Digital images

In digital image processing, images can be represented in several dimensions. The more general form is two dimensional (2D) array formation represented as a mathematical functions in the x-axis and y-axis. Where each specific image coordinate (x,y) represents a given pixel position of the image, and each pixel coordinate has a specific value representing the actual data of the image. As seen the Figure 2 below. Each column represents the x-axis and each row represents the y-axis. When reading the image into the system the point of origin starts from the upper left corner of the image.

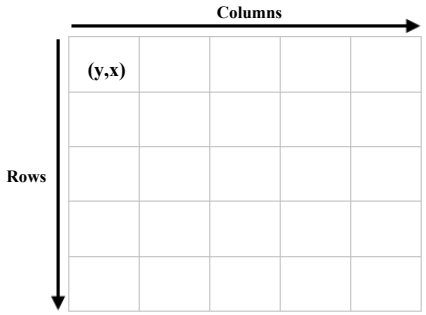


Figure 2: Digital image representation

In reconfigurable devices such as FPGAs, digital images are streamed into the

system sequentially in a method known as a raster scan. Each row of the image is streamed in one pixel every clock cycle, starting from the top left to the bottom right in a zigzag manner as shown in the following Lenna image Figure 3.

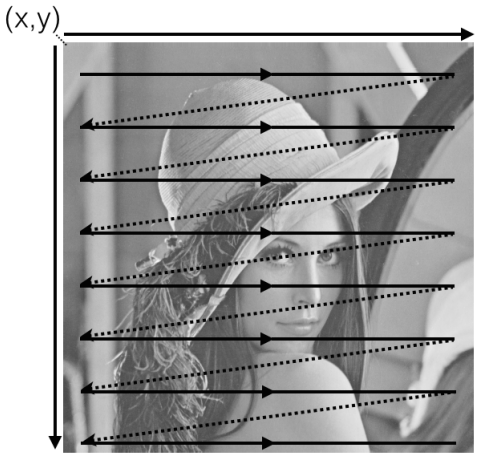


Figure 3: Rastor scan demonstration

Colour model is the grouping and organization of specific colours into a certain colour spectrum. While colour space is a standardized mapping of the real colours into the model's values. Each space allows colors to be represented on a particular designated media such as on displays, screens, printing, etc.. in which each of these media require certain color specifications in order to be represented correctly. Colour models are represented in the form of tuples, such as RGB, Lab, CMYK, and HSV. And each tuple has different values assigned to it representing colours and/or intensities as will be discussed later in this chapter. A three-dimensional representation (also known as colour gamut) of each colour model can be created by representing the main colour groupings such as red, green, blue or cyan, magenta, yellow, and black on a separate axis. Together with each positioning every possible colour variation from this model can be obtained by combining the values of each axis.

Colours can be represented numerically in various ways, such as fractional, percentage, and and integers (ex. 8-bits, 16-bits, 32-bits). The most common method is representing each colour as an 8-bits integer of a finite range that varies between 0 and 255. As such, the values of pure red is (255,0,0), pure green is (0,255,0), and blue is (0,0,255). With the numerical representation of colours it is easy to perform different image processing techniques on digital images.

The RGB colour model stands for the primary colours: red, green, and blue. A vast array of colours is produced when the three colors are added and mixed together in different proportions, hence the name additive colour model. Colours refers to the light beam's wave lengths which are added together to form the different

colour representations. White colours are produced by having maximum intensity value while on the other hand, black colour is produced by having 0 value intensity of each colour component. Equally balanced values of the RGB colours produces a gray colour, where it's shade depends on the how high or low is the intensity level. The following Figure 4 shows the RGB colour model represented as a three-dimensional cube with the different colour variations.

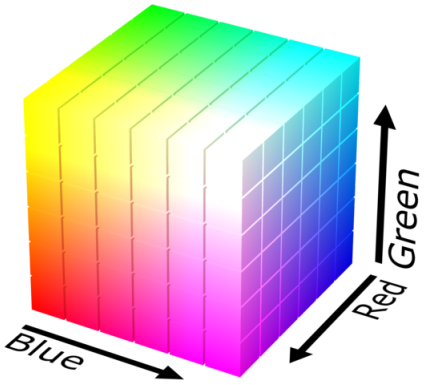


Figure 4: RGB colour model as a 3D cube [9]

The RGB model is mainly used in digital image representations on screen displays such as computers and TVs. This model is based on the human perception of colours that's why it is widely used in photography and display.

The HSV and HSL colour spaces are a transformation of the RGB colour model. HSV is short for hue, saturation, and value, while HSL for hue, saturation, and lightness. Both spaces have proven to be more intuitive, perceptually relevant, as well as light invarient than the standard RGB model [10]. They are represented in the form of a cylindrical coordinates where the hue is the angle around the vertical axis, saturation is the distance from the central axis (differs in value between HSV and HSL), and the value/lightness is the vertical distance along the axis. Grey, neutral and achromatic colours are located along the central vertical access of the cylindrical coordinates. Figure 5 below shows the 3D representation of the HSV/HSL colour spaces.

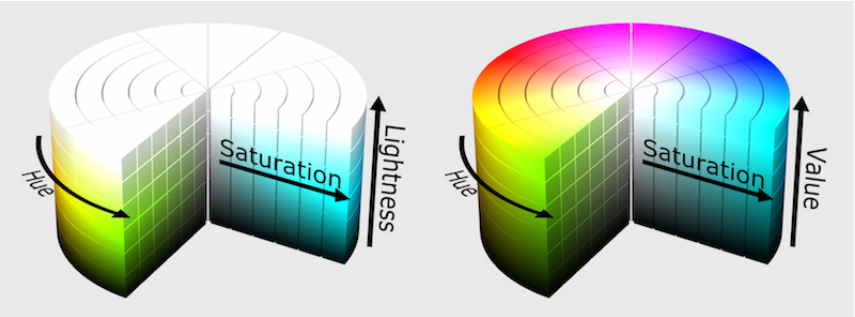


Figure 5: HSL and HSV 3D cylindrical representation [11]

Hue represents the pure colour of the model. And since it is expressed as the angle around the vertical axis, it's values are expressed in degrees and rotates from 0°to 360°. Pure red value get starts from 0°passing through green at 120°and blue at 240°and returning to red again at 360°.

The following Formula(2.1) shows how to calculate hue from the RGB values Hue (H):

$$H = \begin{cases} \frac{G-B}{\Delta} \mod 360 * 60 & \text{if max } = R \\ \frac{B-R}{\Delta} + 120 * 60 \ , & \text{if max } = G \\ \frac{R-G}{\Delta} + 240 * 60 \ , & \text{if max } = B \end{cases} \tag{2.1}$$

Where Δ is the difference between the maximum and minimum values of the RGB tuple.

Saturation represents how white the colour is, meaning pure red, green, and blue colours are considered fully saturated and expressed numerically with the value of 1. Any value below 1 represents tints of the pure colour as well white when the value is 0. In the cylindrical coordinates, saturation is white at the top center and as we move away the saturation value is increased.

Saturation value for both HSV and HSL can be calculated using the below formulas (2.2 and 2.3).

Saturation (S_{HSV}):

$$S_{\text{HSV}} = \begin{cases} 0 \ , & \text{if V } = 0 \\ \frac{\Delta}{V} \ , & \text{if Otherwise} \end{cases} \tag{2.2}$$

Saturation (S_{HSL}):

$$S_{\text{HSL}} = \begin{cases} 0\ , & \text{if L } = 0 \\ \frac{\Delta}{1-|2L-1|}, & \text{if Otherwise} \end{cases} \tag{2.3}$$

Lightness/Value represents how bright or dark the colour is. In HSV a value of 0 is full darkness (black) and a value of 1 is full brightness (white), while in HSL it ranges from 0 to 1/2. This is expressed as moving from the bottom to the top of the cylindrical coordinates.

The methods for calculating Lightness and Value are shown in the following formulas (2.4 and 2.5):

Lightness:

$$L = \frac{1}{3}(R + G + B) \tag{2.4}$$

Value:

$$V = \max\{\text{R,G,B}\} \tag{2.5}$$

HSV colour space has the advantage of being similar to how humans perceive colours. Also colours in the HSV space are light invariant and immune from the different variations of light intensities. This is due to light intensities are already represented in a separate component "Saturation" from pure colours (chroma) "Hue" of the colour space.

2.2.2 Image segmentation

Image segmentation is a group of image processing techniques and algorithms used in computer vision. It segments and partitions a digital image into groups or clusters of pixels that share similar characteristics or features. The purpose of segmentation is to allow easier access to change and manipulate digital images as a first step of further image analysis such as feature/data extraction, pattern recognition, boundaries and object location. Segmentation algorithms falls into two main categories colour-based [12] [13] and shape-based algorithms. In colour-based approach objects in images are segmented into different regions according a pre-specified criteria based on colours. While shape-based algorithms partition images according to the presence of objects inside images.

Thresholding is a basic segmentation technique that converts colour-based or grayscale-based images into binary images. This is performed by looping through the entire subject image and replacing each and every pixel with 0 (for black) if the image intensity is less than or equal a certain predetermined threshold value, or

replacing the pixels with 1 (for White) if the intensity is above the threshold value as shown in equation (2.6) below there are many thresholding algorithms available including colour-based (ex. RGB, HSV thresholding), histogram-based(ex. Otsu thresholding), and local-based (Adaptive thresholding).

$$P_{x,y} = \begin{cases} 0\ , & \text{if } I_{x,y} < T \\ 1\ , & \text{if } I_{x,y} >= T \end{cases} \tag{2.6}$$

Where $P_{x,y}$ is the new binary pixel at x,y coordinates, $I_{x,y}$ is the image pixel at x,y coordinates, and T is the threshold value.

2.2.3 Spatial convolution

In digital image processing, spatial convolution Is one of the most important and frequently used operation for changing the spatial frequency features of an image. Applications of convolution includes and not limited to edge detection, noise reduction, feature detection, smoothing, blurring, sharpening, etc... In image processing convolution can be performed in One or more spatial dimensions, the most popular form is 2D convolution computed for both the horizontal and vertical axis of the image.

Convolution is computed by convolving a source image with a small matrix filter mask sometimes called kernel or convolution matrix. In most cases, the mask is square shaped and has an odd number size of 3x3, 5x5, 7x7, 9x9, etc... The center pixel of the mask is called the pixel anchor point (sometimes the anchor pixel can be appointed to the upper left corner cell of the mask). Theoretically, the mask is placed on top of the image where the anchor point of the kernel is positioned on the source pixel as shown in figure 6 below. Accordingly, the convolution is computed and the result is stored as a new pixel in a newly formed image container.

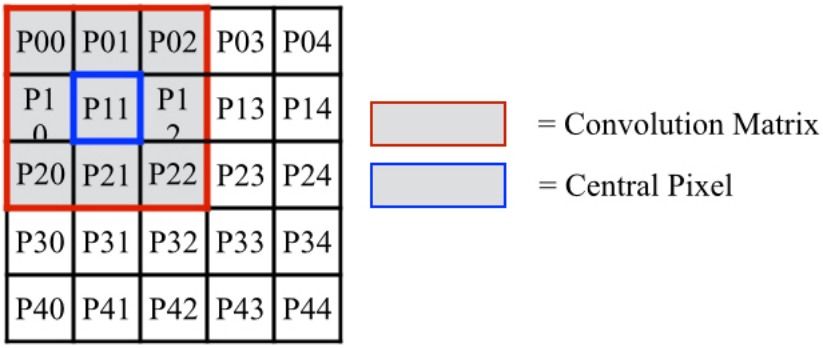


Figure 6: Performing 2D convolution on an image

2.2.4 Noise reduction

In most cases images captured by digital camera sensors are prone to include additional inferior signals referred to as noise. There are many types of noise that can be produced during image capturing and/or processing; for example salt and pepper noise, pixels of different colour intensities from there neighboring pixels, shot noise, and periodic noise. Usually these noise appears on images as tiny grains, light and dark sport, dark corners, uniform horizontal/vertical periodic bars, or sometimes appears as an extension to the image objects. In addition, unwanted data information can be produced from a certain image processing algorithm which is not needed for later stages of the system processing. In order to remove inferior data from images, various algorithms and techniques was developed to tackle the reduction and/or removal of different types of noise. This includes linear smoothing filters, nonlinear filters, non-local means, and wavelet transform.

Noise reduction needs high computational power and resources to perform well, depending on the algorithm used. Also one of the trade-offs of using noise reduction algorithms is that they tend to remove important information, distort, and smooth out the original subject image. In that case certain precautionary measures should be taken into account to minimize the effect of those trade-offs.

2.2.4.1 Median filter

A nonlinear noise reduction filter usually used as a preprocessing step for improving results in further processing stages. The filter is widely used in image processing since it is characterized by preserving the edges of image objects during the noise reduction process. The median filter replaces every pixel of the source image by the median value of the pixel's neighboring values. The strength of the noise reduction process depends on the kernel size used to calculate the median value of the neighboring pixels. The filter kernel act a sliding boundary window which limits the number of pixels utilized to calculate the median of the anchor pixel.

2.2.4.2 Morphological operators

A nonlinear analysis and processing operations based on the morphology or shape of digital images. The operators were originally designed to be executed on binary images, but it was extended later on to operate on grayscale images. The operator's calculation technique does not depend on the numerical values of pixels, but on the relative ordering of the image pixels, hence the reason it works well on binary images. The main purpose of the operators is to enhance the build and structure of images, also the operators can be used as a method for noise reduction or the removing unwanted objects or imperfections in images. The most common algorithms for morphological operators are erosion, dilation, opening, and closing.

The morphological operators slides over the source image with a small window called structuring element. Every time the structural windows slides over the image the operator of choice checks if the values of the structuring elements fits within the neighborhood pixels of the image or hits/intersects with the image neighborhood pixels. The figure 22 below shows the basic idea of the "fit" and "hit" concept. In binary images, operations that are indicated as a "fit" occurs when every pixel of the structuring element is 1 as well as the corresponding pixel on the source image. A "hit" is said to occur when at least one pixel is set to 1 in the structuring element and the corresponding pixels in the source image is 1.

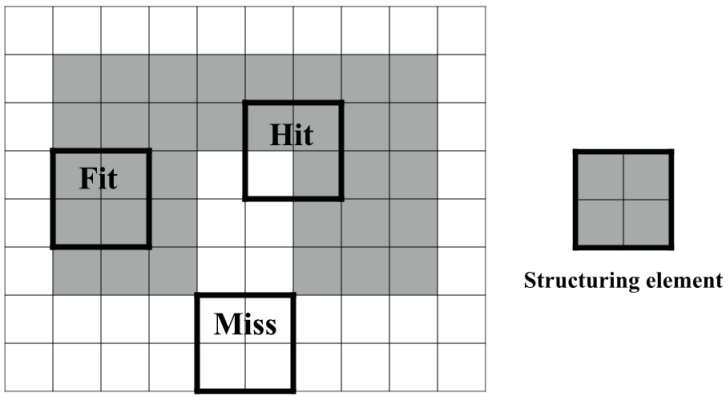


Figure 7: Morphological "Fit" and "Hit" executed on an image

The structuring element is a small window of a specific size specified by the matrix dimension (ex. 3x3, 5x5, etc...) and, shape represented as the patterns of the values of ones and zeros inside the matrix (the most common shapes are diamond, cross, and rectangle). The origin of the structuring element or the anchor cell usually is located at the center of the matrix, also it can be located at any point inside or outside the window.

Erosion technique is one of the available morphological operators, is the process of eroding pixels from the source image, the erosion is represented by the following formula (2.7).

$$g = f \ominus s \tag{2.7}$$

Where g is the output binary image, s is the structuring element, and f is the source image. The method slides the structuring element over the source image and attempt to find a "fit" in the source image, if successful the corresponding pixel of the output image is set to 1 otherwise is set to 0. Erosion removes thin strip of pixels from the outer and the inner boundaries of the area, it also remove small details when larger structuring elements is used.

Dilation is the opposite process of erosion, is the process of dilating or adding pixels to an image where thin layers of pixels is added to the inner and outer boundary of the area. The dilation formula (2.8) is shown below.

$$g = f \oplus s \tag{2.8}$$

The structuring element slides over the source image and attempt to find a "hit" in the source image, if successful the corresponding pixel of the output image is set to 1 otherwise is set to 0. Dilation is quite often used to fill in small gaps/holes or disconnected objects present in images.

Opening and closing is another form of the morphological operators based on a combination of algorithms between erosion and dilation to achieve different effect depending on the application. In opening, the operation performs erosion followed by dilation. It is used to open gaps between connected objects in an image. While in closing, the operation is executed by performing dilation followed by erosion. Closing fills in small gaps between objects while maintaining the original size of the object.

2.3 Reconfigurable integrated circuits

Reconfigurable integrated circuits is an advance computer architecture circuitry that combines the high performance and reconfigurability of hardware, with the flexibility and availability of software [7]. Reconfigurable circuits have the ability to process large amount of data within a short time using high performance logic fabric circuits, as contrast to general purpose microprocessors where the processing speed is limited to the processor's capabilities. One great feature about reconfigurable devices is their capability to reconfigure and customize the design at runtime without the need to send the new design to the fabrication lab, such as in the case of Application Specific Integrated Circuits (AISCs). At any point in time a new design or alteration to a design can be easily completed and flashed to the reconfigurable device in a short time. Hence, reconfigurable devices is very popular in certain highly critical domains such as defense and space that requires real-time, high precession, and high performance results.

3 State of the Art

In this chapter, the latest research and ideas produced in image processing will be discussed, specifically in the area of image detection and classification. Moreover, details about reconfigurable hardware device utilized in this thesis project will be discussed. A demonstration of the characteristics, behavior, and architecture of the Field Programmable Gate Array (FPGA) together with it's main advantage, parallelism will be detailed later in this chapter.

3.1 Feature extraction

In computer vision and machine learning, feature extraction is the process of detecting and extracting data of a certain characteristics or features from an image or an algorithm. Generally, any large data source such as images and algorithms have the tendency to be redundant. Hence, reducing these large blocks into smaller sets of similar features will enable efficient and fast extraction of relevant data which is required by the task. Feature extraction overcome the universal problem of having limited number of memory resources as well as having expensive computational power to analysis large amount of data.

In image processing, feature extraction is used to extract certain features and shapes of objects present in digital images and videos. Feature extraction are used for various application for example edge detection, scale-invariant feature transform, blob/circle detection, and corner detection. Algorithms such as template matching, hough transform (with all its variations), and blob extraction are widely used techniques to extract features from digital media.

3.1.1 Template matching

Template matching is a popular computer vision feature extraction and object identification technique. The technique is based on matching a template or a group of templates to an image or a video stream [12]. A small section of the image containing the region of interest is selected from the source image and extracted into a new template template [14]. The center pixel of the template which is mostly in the form of a matrix containing pixel values of the template image is positioned on top of the source image pixels. Next, a similarity matching algorithm is applied for the complete image until every image pixel generates a matching result. Results

produced from the matching algorithm are stored iteratively in an accumulator matrix for later analysis. Finally, an algorithm is applied to the accumulator to check the highest point score indicating the best available position of the object in the image. In addition to finding the object, template matching algorithm stores the (x, y) coordinate locations of the matched object.

Template Matching methods are straightforward however, they are relatively computationally expensive. The algorithms needs lot of computational power, memory resources, and time to perform the computations specially when performing the operation on large high-resolution images and videos [14].

3.1.1.1 Cross-Correlation (CC)

Cross-correlation is a procedure that measures the similarity between two signal functions, also known as sliding dot product. In most cases a shorter signal *(g)* is applied to find a similarity in the longer signal *(f)*. Cross correlation is used in image processing for feature extraction and pattern recognition particularly if the application does not require the algorithm to be light-invariant to the different conditions of light.

Cross-Correlation technique is not robust since it is non-invariant to light, as results obtained from CC changes greatly according to the current lightning and illumination condition of the subject media which makes it a poor choice as a similarity matching technique for images.

3.1.1.2 Normalized Cross-Correlation (NCC)

Normalized Cross-Correlation (NCC) calculation methods are used in template matching methods which uses pixel based comparisons to detect the similarities between images and templates [15]. Like other similarity , NCC is used mainly when the brightness and light illumination of both the template and source image differs in different lightning conditions. The approach normalizes the cross-correlation by subtracting mean value of the image in relation to the standard deviation, as a result of the normalization, this makes the NCC invariant to light. Several papers introduced a modified version of NCC called fast normalized cross-correlation [16]. The new modification simplifies the calculations of the denominator and the numerator of the NCC coefficient hence, reducing the computational power of the calculations. One paper suggested a new approach for NCC denominator calculations [15], the new method relies on calculating sum tables containing the integrals for each image. This approach reduces the order of magnitude when compared to traditional Fast Fourier transform method.

3.1.1.3 Convolution

Convolution is another approach to perform template matching. Similar to correlation, it is a mathematical operation performed between 2 discrete functions $f(x)$ as

the image function and $g(x)$ as the template filter [32] [31]. It is computed based on the sum of products between the two coefficients. Each convolution result is computed by iterating through the entire image, the highest output value indicates a possible match. The following equation (3.1) shows the formula for performing convolution on 2 discrete signals.

$$f[x,y] * g[x,y] = \sum_{j=0}^{j-1} \sum_{k=0}^{k-1} f[j,k].g[x-j, y-k] \tag{3.1}$$

Performing template matching using spatial convolution as the matching method is easy to implement, however this algorithm is slow and not optimal as it still requires lot of resources more computational power specifically when using large images.

3.1.2 Circle Hough Transform (CHT)

Circle Hough Transform (CHT) a variation of the popular Hough Transform (HT) shape detection algorithm [17]. CHT searches for circles in an image and for each possible match results are stored in an accumulator matrix where a voting operation is executed to find the local maximal representing the located circle. The algorithm depends mainly on calculating the parameter and the x and y locations of the center of the circle. Meaning, performing computations for 3 parameters, increasing in the computational complexity of the process. Moreover, the calculation method becomes very expensive in terms of memory resource allocation and for that reason, 3 dimensional CHT is less used in hardware development.

However, several papers proposed more efficient and less complex approaches to CHT. In their research J. Illingworth and J. Kittler introduced the usage of a small accumulator matrix by using what is called by "flexible iterative accumulation and search strategy" to obtain distinguishable peaks in the parameter space [18]. Another research by Xin Zhou, Yasuaki Ito, and Koji Nakano introduced the concept of using only one dimension for the parameter space calculation instead of three [17]. Here the (x, y) and r will be detected in series. And for each detection only 1D parameter space will be used and then passed over to the next variable when detection of the previous is complete. This method reduces the need for large number of storage resources as well as reducing the computational complexity of the calculations.

3.1.3 Fast radial symmetry transform (FRS)

A new point of interest detection algorithm introduced by G. Loy and A. Zelinsky in their paper [19]. The transform detects points of interests in images based on high radial symmetry. Although similar to CHT, this approach focuses on how each image pixel contributes to the surrounding pixels instead of the local neighborhood

contribution to the central pixel.

The transform uses a voting mechanism similar to the one used in Hough Transform to detect the points of the local radial symmetry. The transform uses a one dimensional parameter space and an accumulator for the computations. This approach used Gaussian filter in one stage which increase the computational complexities. Another approach [20] suggested the removal of the Gaussian smoothing filter to reduce the computational complexities of the algorithm. FRS provides lesser computational cost as compared to other algorithms such as CHT, also it delivers quite acceptable results in terms of performance. One drawback of this approach, that the algorithm is not invariant to noise and distortions which could be problematic if used for low-resolution noise prone images.

3.2 Field Programmable Gate Array (FPGA) Architecture

Field Programmable Gate Arrays (FPGAs) are integrated circuits reconfigurable devices also known as programmable logic. The architecture of a typical FPGA is characterized by combining the flexibility of software with the high performance of hardware. FPGAs provides circuitry with high power, high performance, and logic area in contrary to general purpose software architectures [7]. In addition, FPGAs allows reconfigurable designs in the sense that, systems can be redesigned, developed, and launched into the device with ease, anytime and as many times as needed. Like other Very Large Scale Integration (VLSI) reconfigurable hardware circuits, FPGA devices have a key characteristic of supporting parallelization via pipelining the application into stages to deliver high performance results. FPGAs are very useful when designing complex systems which requires processing of large amount of data within a small amount of time yielding high performance, accurate results. Accordingly, FPGAs are widely used in real-time embedded systems in domains such as defense, space, medicine, and manufacturing industries.

FPGA hardware accelerators was originally planned to be a generic programmable devices for general purpose computer systems since floating point processing elements present in FPGAs was known to accelerate numerical and arithmetical operations successfully with high speed. Later on, with the rapid advancement in technology in recent years FPGa devices may contain sophisticated hybrid co-processor CPU elements on board of the device hence, integrating software and hardware processing elements together. This arrangement known as System on Chip (SoC) FPGAs, SoC devices such as the Xilinx Zynq 7000 EPP is characterized by having high bandwidth for FPGA-CPU communication, low power, small board size, integrated memories and transceivers and an array of peripherals. This blend of hardware/software co-design produces extreme real-time processing capabilities with

high performance and robust results.

The main building bock of an FPGA mesh system architecture consists of Configurable Logic Blocks (CLB) surrounded by input/output blocks, and connected with programmable Interconnect. A typical FPGA is composed of tens of thousands of logic blocks, flip-flops and memories. On the other hand, not every programmable interconnect is connected to every logic block directly, FPGAs provides the capability to place and route logic on the device through connection blocks and switches. Figure 8 shows an abstract view of a generic FPGA mesh architecture.

Despite delivering results with high speed and performance, the nature of reconfigurable computing comes with a cost of complexity. FPGA developers need to work with both software and hardware in which each has their own unique design techniques and way of thinking. Designing in software typically is sequential while in hardware concurrency and parallelism is the main focus hence, shifting from SW to HW design may be confusing and complex at sometimes.

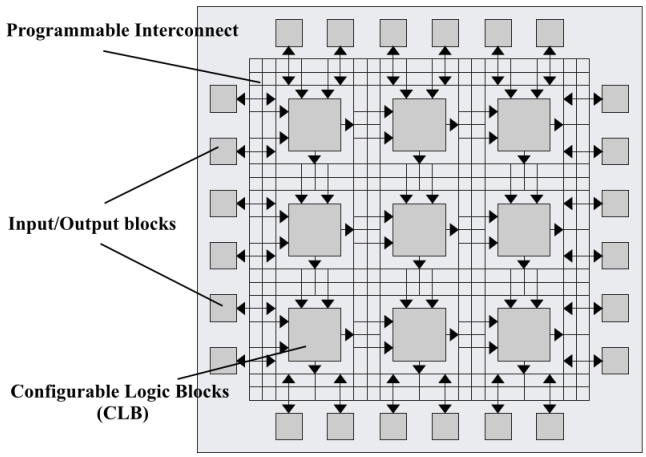


Figure 8: An abstract view of basic FPGA architecture [1]

3.2.1 Fine-grained and coarse-grained FPGA programmable fabrics

There are two main classes of FPGAs programmable fabrics, fine-grained and coarse-grained fabric blocks. The fine-grained blocks are found in conventional FPGAs, where the devices have relatively huge number of simple CLBs which is determined based on the technology used to develop the target family.

In recent years, scientists are attempting to create optimized FPGA devices to reduce critical paths, area size, delay, and power consumption. In attempt to achieve that, large amount of studies concentrated on developing FPGA devices which em-

bed and integrate coarse-grained block elements into the conventional fine-grained fabric. These hybrid FPGA designs have large logic blocks which incorporate embedded arithmetic multipliers and processors which increase the speed and efficiency of the implemented functions. One drawback of coarse-grained blocks is that, if the blocks was not fully utilized by the application, the architecture waste area accordingly.

When designing hybrid FPGA architectures, it is important to implement somehow flexible routing blocks or structures between the coarse-grained blocks and fine-grained elements in oder to guarantee routable designs. If the routing blocks are very flexible, the routing elements will consume large area and will be slower.

3.2.2 Configurable Logic Blocks (CLB)

The CLB, is an integral component of the FPGA circuitry, it is the building block that include one or more slices. Depending on the fabric architecture of the FPGA, each CLB slice is composed of a finite number of flip-flops, Look-up tables (LUT), and multiplexers used for implementing combinational and sequential logic. Depending on the FPGA family, the composition of the slices can differ. One example, Xilinx Virtex-2 FPGA has two flip-flops and two LUTs, while Xilinx Virtex-6 FPGA has eight flip-flops and four LUTs. Some slices of certain FPGA families support the utilization of distributed RAM and data-shift. In that case, these slices are called SLICEM and SLICEL respectively.

Each CLB is connected to a switch matrix that acts as a routing channel to connect with the general programmable interconnect. In addition to I/O interfaces for connection with external source signals. With this complete network, the CLB will be able to perform complex logic operations, synchronize programmed functions, and implement memories on the FPGA. The figure 9 below shows an example of the Xilinx Virtex-6 slice arrangement inside a CLB.

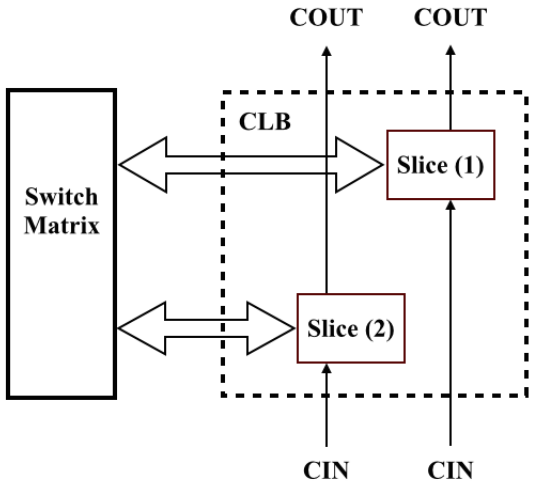


Figure 9: Xilinx Virtex-6 slice arrangement inside a CLB [21]

3.2.3 FPGA programmable routing network

The programmable routing network also known as the programmable interconnect is a logic fabric that designed to establish routing connections and communication between the different CLBs and I/O blocks to implement and describe user defined circuitry. The routing network is composed of programmable switches which can configured as well as wires for the physical connection of the network.

In a mesh-based FPGA architecture the CLBs are located in a grid or a mesh of programmable interconnects as illustrated previously in figure 8. Switch Boxes (SB) are used to connect the horizontal and vertical routing wirings. Connection Boxes (CB) are used to connect each CLB with the routing network. Figure 10 shows an illustration for the a switch box and a connection box provided by a typical mesh-based FPGA architecture.

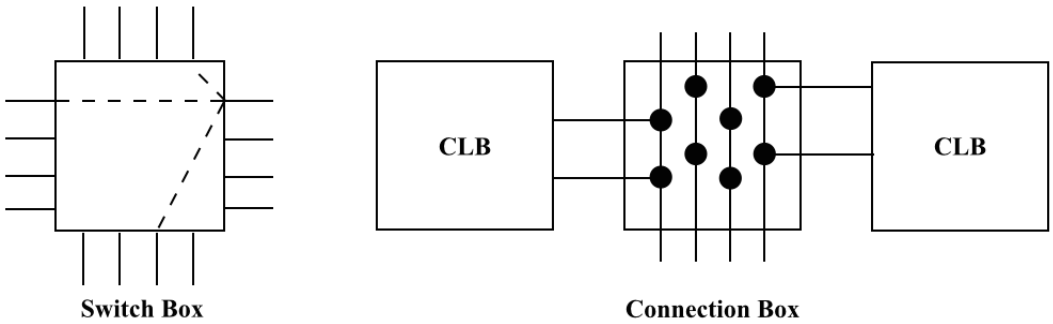


Figure 10: Switch box and connection box illustrative figure

3.2.4 Memory

Storage units are very important elements in any FPGA design. And since they are scarce particularly on-chip storage, hardware systems must be an designed in an optimum and efficient way to utilize adequately the memory resources on the target device. There are two types of storage elements In FPGA devices, on-chip and off-chip memories. There are two on-chip memories located on the an FPGA chip Block memory and distributed memory. The Block RAMs is distributed to the left and right sides of the chip. Block RAMs cannot be used to execute digital logic functions, but only to store data. Depending on the target device family and vendor the size of the Block RAM can differ. For example Xilinx Spartan-6 devices have block RAMs of size up to 4,824 K-bits, while Xilinx Virtex-6 device family has a size up to 38,304 K-bits. Other type of on-chip memories are distributed memories, located inside each CLB, they have single or dual port RAM in the form of LUTs. Distributed RAMs sizes are smaller than Block RAM's for example in Xilinx Virtex-6 devices, each CLB has a size up to 6,370 k-bits of distributed RAM. And for that reason, distributed RAMs are mainly used for small size memories to create simple registers, FIFOs, and scratch-pad memories. On the other side, block RAMs are used for memories of a larger size.

In cases when complex designs that requires lot of computations using on-chip storage will be very limited. However, Off-chip memories, in the form of static RAM (SRAM) or dynamic RAM (DRAM) becomes necessary. Off-chip RAMs provides high bandwidth, low latency, and greater storage capacity than on-chip RAMs.

3.2.5 Hardware Description Language (HDL)

FPGA logic designs are implemented in the form of a hardware description language (HDL) like VHDL (Very High Speed Integrated Circuit Hardware Description Language) and Verilog. HDL describes the structure and behavior of the digital logic with a very high precision in a very abstract level. Like regular programming languages, HDL has expressions, operators, conditional statements, input/ outputs, processes, and functions. When compiled and synthesized the output of the HDL is a gate map in the form of a net-list, the following section will discuss design flow in details.

3.2.6 FPGA synthesis design flow

An FPGA synthesis design flow is a straight forward procedure. It's main purpose is to optimize the logic design for maximum performance and to ensure that the converted logic fits well on the selected target FPGA device. The optimization process passes through six stages from logic synthesis, technology mapping, packing, placement, routing, and finally bit-stream generation. The output bit-stream is then loaded into the FPGA device for the final execution of the design. Figure 11

illustrates the full optimization flow of a typical FPGA design.

Before starting the optimization process, the system first is designed and described in a hardware descriptive language (HDL) and fully tested using one of the available simulation tools for verification. The optimization process starts with "Logic Synthesis" which converts the HDL code (behavioral and structural) into logic design. Then, "Technology Mapping", maps the logic (gates) design to k-bounded LUTs. Next, the "Packing" stage also known as clustering, group the k-bounded LUTs as well as the available flip-flops and form k-logic blocks (LB). This stage also, groups the k-LBs together to form LB clusters which is then mapped directly into the FPGA logic elements. Next two stages are the "Placement" and "Routing", here the mapped LB clusters from previous stage is placed and routed into the it's designated LBs on the FPGA. Finally, the "Bitstream Generation", generates a binary file that establishes all FPGA programming procedures to configure the all the routing, CLBs, I/O, and memory resources on the FPGA. Once this is done, the bitstream binary file is loaded into the FPGA and the designed logic circuit is set to be ready for operation.

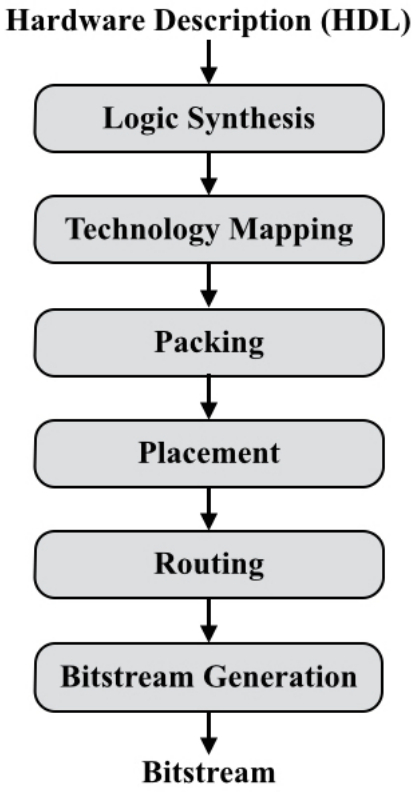


Figure 11: Typical FPGA design flow

3.2.7 Parallelism and pipelining

FPGA devices have a unique characteristics as it combine the flexibility of general purpose softwares with the high performance and efficiency of hardware. Since designing for hardware is very different from designing for software, in the traditional software architectures instructions are executed sequentially this is due to the sequential nature of the traditional microprocessor architectures. On the contrary, hardware design for FPGAs have the advantage of possessing a large number of logic gates and elements that can be described using HDL to operate different tasks and operations in parallel. In order for FPGAs to outperform general purpose software-based processors, the design must exploit greatly the advantages of parallelism. As stated in [7], task parallelism divides the hardware design into subsections executed and processed in parallel. Where each subsection is fed with data to perform separate processing tasks required by the module, finally the outputs are grouped together and analyzed for further processing. This technique speed up the processing times of the system yielding high performance results.

Pipelining is a technique performed during digital design to increase the rate of parallelism and avoid data dependencies in hardware design. During the design process of a particular functionality, the operation is split down into several stages in which all the stages are executed in parallel within the same clock cycle. Outputs obtained from each computation at each stage, are received by the following stage (outputs from the previous clock cycle) for further computation.

The following image 12 shows a blocked diagram of a pipelined architecture using registers designed for reconfigurable devices of a simple example mathematical function as shown below in (3.2) . Here by adding intermediate registers to the pipelining operation reduces the total computational latency of the whole operation, as they allow all the intermediate stages and data paths to run in parallel within the same clock cycle.

$$z = (a * y) + b + c \tag{3.2}$$

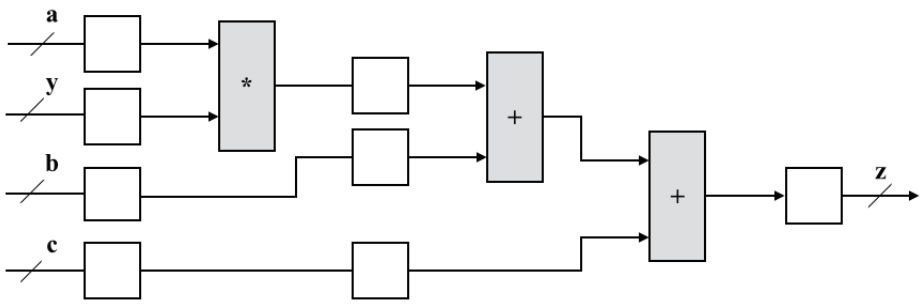


Figure 12: Pipelined architecture example

4 Traffic Sign Detection System

In this chapter the proposed system architecture and implementation of the traffic sign detection system will be discussed in details illustrating the main functionalities and purpose of each module. In the following figure 13 the complete system design is shown for both the software and hardware architectures. The system is composed of four main functional components the preprocessing module, the segmentation module, merger module, and the detection module.

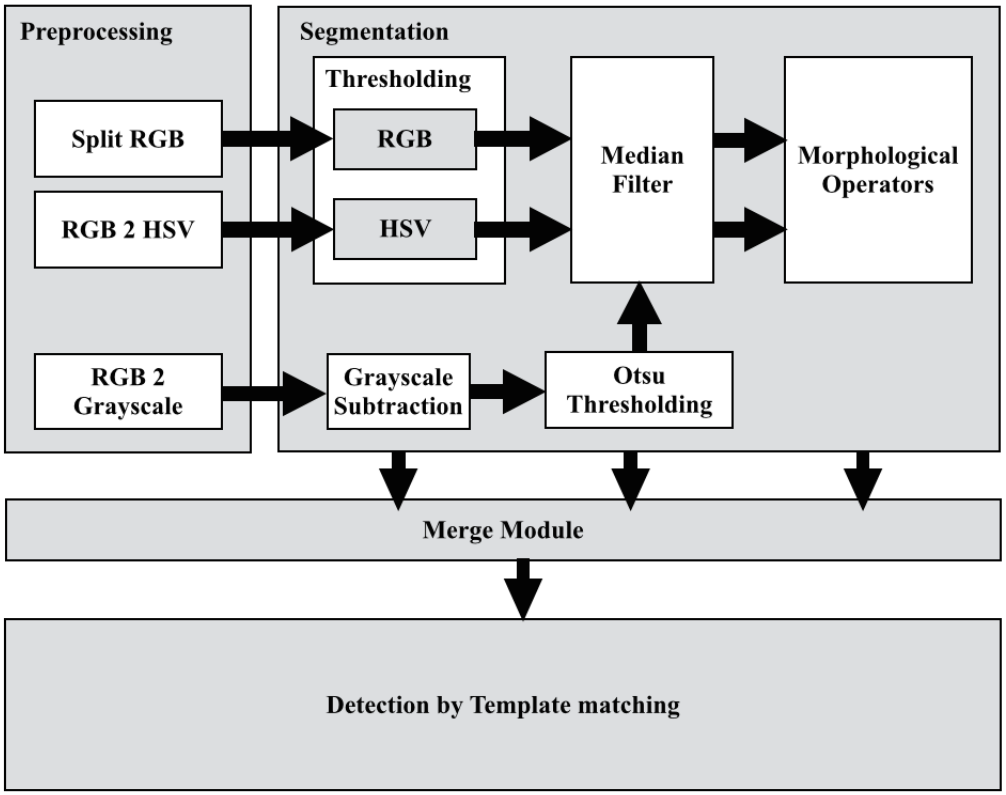


Figure 13: Block diagram of the proposed traffic sign detection system

The main purpose of the preprocessing module is to prepare the needed data ahead which will facilitate further processing of the image in later stages. The preprocessing module has 3 independent units each performing different task and each executing in parallel. The "Split RGB" unit, splits or extracts the RGB channels of the input image into separate pipelined registers. The "RGB conversion" unit that

has the "RGB 2 HSV" unit which converts the RGB colour model into the HSV colour space. And the "RGB 2 Grayscale" unit which converts the 24-bits RGB into an 8-bits Grayscale.

The Segmentation module is grouped into 5 main components which execute different segmentation and noise reduction algorithms. The module extracts the required region of interest from the source image as well as produce images free of noise and impurities which might occur in images. The segmentation components are: colour-based thresholding, Otsu thresholding, Grayscale subtraction, noise reduction, and morphological operators. The module produces three binary output each produced from one of the three segmentation methods used in the system.

The Thresholding component in the Segmentation module receives two outputs from the preprocessing module, the three 24-bit channels of the RGB image and the three 24-bits channels of the HSV colour space. In addition, the Grayscale Subtraction component receives the transformed 8-bit Grayscale image. output from the Segmentation module. Once the data is received by the segmentation module, the Threshold and Grayscale Subtraction components starts executing in parallel to achieve maximum efficiency.

The output of the Grayscale Subtraction is then forwarded to the Otsu thresholding to convert into a binary image. Later on, the results from the colour thresholding and the Otsu thresholding components are passed to the median filter and then to the morphological operators for noise reduction and elimination of impurities in the output binary images.

Since individual execution of segmentation techniques may produce in an incomplete or unclear desired results, it was decided to merge the outputs of those algorithms into one binary image to produce better enhanced and robust version of the segment source image. The Merge module receives three binary images from the segmentation module and merge them with an bitwise AND to produce one binary image of the segmented source .

Finally, the Detection module receives the merged binary image from the previous module and performs Template Matching using five templates (in different sizes) of the extracted traffic sign. The output of this module is the best matching result with the x and y coordinates of the proposed location of the detected sign in the source image. The details of the template matching operation will be discussed later in details in the implementation chapter.

4.1 Implementation Overview

The German speed limit signs have distinctive features that makes them very easy to be detected by TSR systems. Speed limit signs have a red circle boundary of a specified diameter and thickness, a white plain background, and a black number centered in the middle of the sign indicating the maximum speed limit on the street. Accordingly, implementing the system using selected image processing techniques and algorithms can yield results with high success rates.

The proposed traffic sign detection system uses the same architectural blocks (Preprocessing, Segmentation, Merge, and Template matching) for both the software and hardware implementations. However, the practical implementation approaches and techniques differs for each platform. This is due to the different hardware architecture nature of both the software and hardware.

This chapter describes two implementation phases of the proposed system. First, the system was implemented sequentially in a conventional software development environment platform. This phase is very important as it serve as a prototype of the proposed detection system. It Gives an overview of the whole system and how the selected image processing algorithms works and reacts when integrated together to form the structure of the detection system. In addition, the software implementation acts as a simulation to validate and verify the correctness and accuracy of the system functionalities. Finally, the software implementation allows the output results of the finished system to be visualized via OpenCV library. The second phase is the main focus of this thesis, implementing a fully parallelized and pipelined hardware acceleration solution targeting Xilinx Virtex-6 FPGA board. This implementation take advantage of the flexibility and parallelism of FPGA hardware architectures to deliver high performance, real-time robust results.

4.2 Software Implementation

The software system is implemented in Eclipse Mars IDE environment using C++ with OpenCV library. The selected image processing algorithms for the proposed system was developed and designed from scratch without using the OpenCV own function algorithms. The reason behind not using OpenCV supported image processing algorithms, is designing the algorithms manually will allow me to facilitate and transfer of these algorithms later on, and to be able to adapt them easily to the hardware platform.

4.2.1 Overview of Software Design Flow

As discussed in the previous chapter (chapter 4), the system architecture shown in figure 13 is composed of 4 main components. The preprocessing, segmentation,

merge, and template matching modules. The software architecture is a little bit different, since there is one additional module developed to reduce the size of the high resolution images, the "Downsizing Module". The module is located between the "Merge" and the "Template matching" modules. Additional details regarding this module will be discussed in a later section.

The following figure 14 illustrates an abstract view of the design flow of the software implementation. The process starts with reading an image from the PC harddrive and forward it to the preprocessing module, next the output results is then forwarded to the segmentation module for further analysis producing 3 output binary results which is fed into the merge module to create one binary image. Next, the binary image is downsized three times before finally fed into the template matching module for the final detection phase. The template matching module will output a new image with a calculated matching location of the detected sign which will then be displayed visually using OpenCV library.

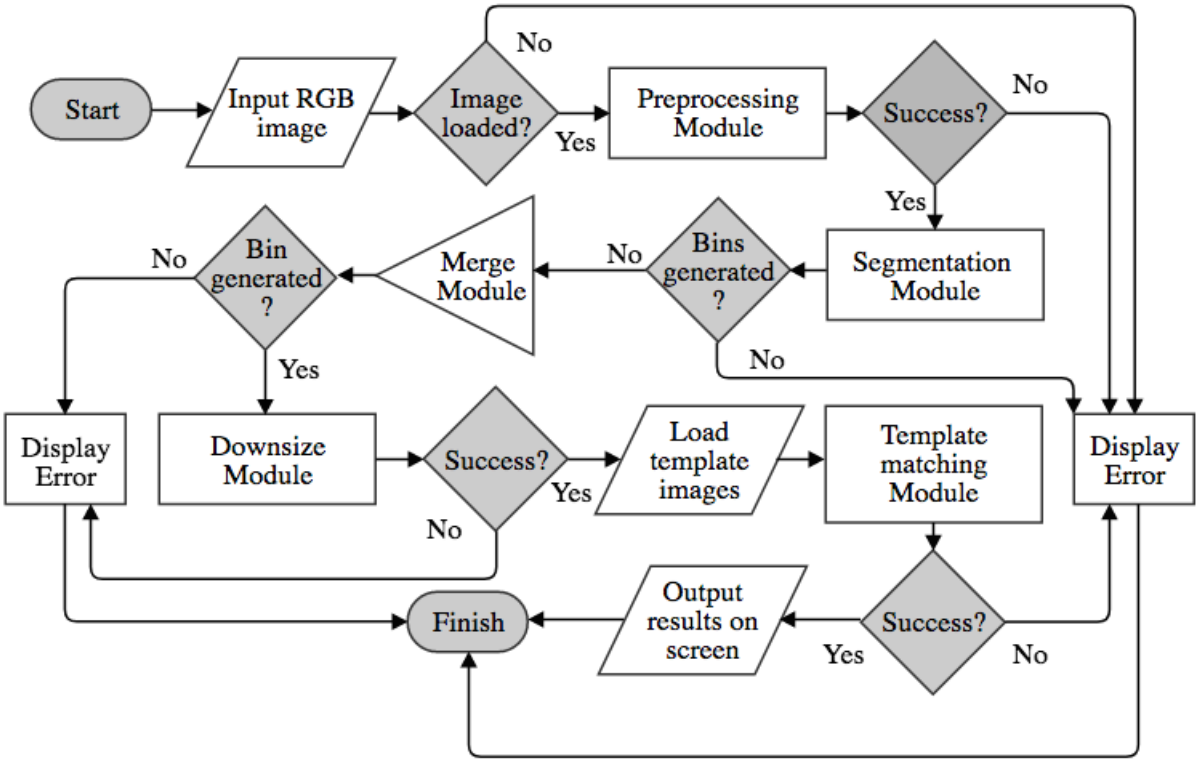


Figure 14: Abstract view of the software design flow-chart

4.2.2 Preprocessing Module

The preprocessing module has three functions represented by "splitImage" and "rgb2hsv", and "rgb2grayscale". Each of the functions execute different task, and

are executed sequentially according to the specific function call in the algorithm. The preprocessing functions are called directly inside the segmentation module at a specific module location. The naming of the "preprocessing module" is solely an abstract representation to indicate a specific operations required before the actual execution of the image processing algorithms.

The "splitImage" function, splits the RGB input image (24-bits channels) into it's three primary colours the red, green, and blue. And with each split the function generates separate image container for each of the 8-bits channels using OpenCV's "Mat" container. As discussed previously, a 24-bits RGB image is typically composed of a large matrix, in which each element designates an image pixel with values ranges from 0 to 255. Since this is a colour image of 3 channels (Red, Green, Blue), each channel is 8-bits wide. For every pixel position in the matrix there are 8-bits x 3 channels = 24-bits. In addition, each pixel value is arranged according to a specific channel order for example RGB or BGR. Pixel position inside the matrix can be easily accessed and the individual channels can be easily extracted based on the predetermined order. In OpenCV the library typically reads colour images in the BGR order, hence for each pixel location in the image accessing the first 8-bit will yield blue value, the second 8-bits yields green value, and the third 8-bits yields the red value. For a typical 8-bits image arrangement, the values for each pixel have a range between 0 and 255. Figure 15 below illustrates the generated channels from the split function.

Figure 15: Red (a), Green (b), Blue (c) channels generated from the SplitImage function

The "rgb2hsv" function converts the RGB colour model into HSV colour space. First, the function calls the "splitImage" to access the extracted R,G,B channels for further processing. Previously in chapter 2, I have discussed the mathematical operation of transforming RGB into HSV shown in equations 2.1, 2.2, and 2.5. As a preprocessing step all values of the H,S,V channels must be normalized between (0-255). As this process will later facilitates the thresholding operation in the segmentation module. As such, the formulas stated above was changed to accommodate this requirement. The following equations 4.1 and 4.2 illustrates the normalized

formulas for calculating hue and saturation. There was no need to normalize the "value" since it is the maximum weigh of the R,G,B channels.

Hue (H):

$$H = \begin{cases} \frac{G-B}{\Delta} \mod 255 * 43 & \text{if max } = R \\ \frac{B-R}{\Delta} + 85 * 43 \ , & \text{if max } = G \\ \frac{R-G}{\Delta} + 170 * 43 \ , & \text{if max } = B \end{cases} \tag{4.1}$$

Saturation (S):

$$S = \begin{cases} 0 \ , & \text{if V } = 0 \\ \frac{\Delta}{V} * 255 \ , & \text{if otherwise} \end{cases} \tag{4.2}$$

The Normalization process for hue is done by dividing the values 120, 240, and 60 by 360 and then multiply the result by 255. As for saturation the output values are multiplied directly by 255.

After the computations the "rgb2hsv" function creates a new HSV image and pass it to the segmentation module for further processing. The following figure 16 demonstrates the different HSV channels (a-c) produced along with the combined HSV representation (d).

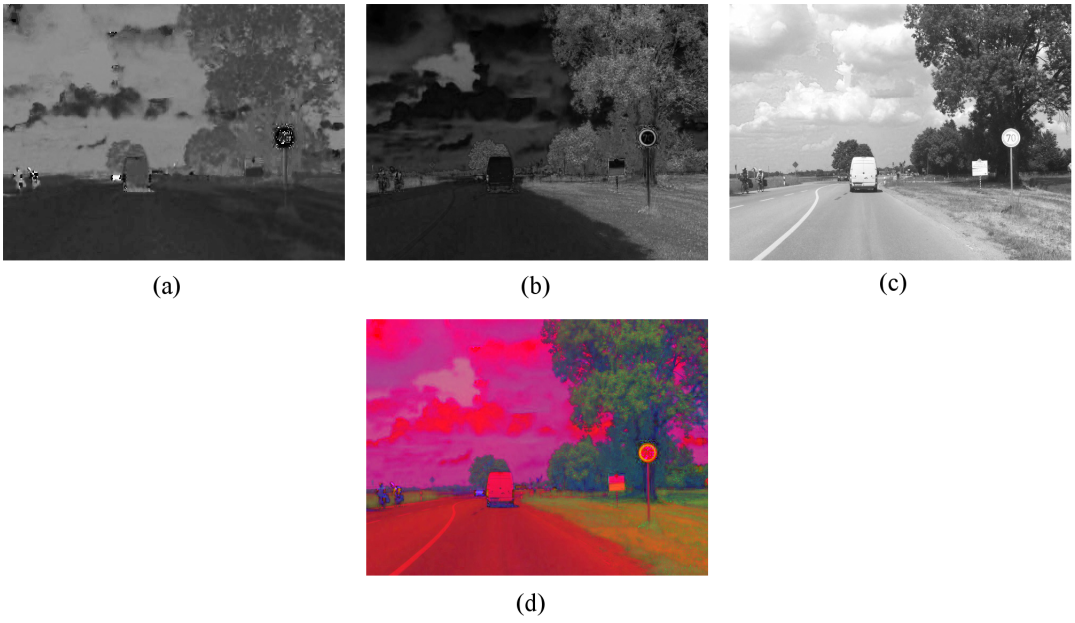


Figure 16: (a) Hue, (b) Saturation, (c) Value, and (d) HSV images generated from the rgb2hsv function

The last function in the preprocessing stage is the "rgb2grayscale" conversion, and for this I used the built in OpenCV functionality to perform the conversion and create the new gray-scale image.

4.2.3 Segmentation Module

The segmentation module perform the core computations of the traffic sign detection system. First it perform sequentially image segmentation of the RGB, HSV, and Grayscale images followed by noise reduction and morphological operations. The software module is composed of seven functions: the "hsvThresh", "rgbThresh", "graysub", "otsu", "medianfiler", "erode", and "dilate" functions. The following sections will discuss in details the operations of each function.

4.2.3.1 HSV Thresholding Function

The HSV thresholding function extracts the H,S,V channels from the HSV image passed on from the "rgb2hsv" function. After that thresholding according to specific values is performed on the individual HSV channels to generate a binary image. The thresholding technique is based on eliminating all colour channels from the image except the red channel by. As a result, the binary image will include only the region of interest and in that case a white circle of the speed limit sign.

Thresholding was performed for not only for the hue, but also for saturation and value. The reason for doing the operation on all three channels increases the chance of producing better, robust and light invariant results than just performing the operation on only hue. Values selected for thresholding was generated based on a "trial and error" bases. Initial values was obtained from a research paper written by Jim Torresen, Jorgen W. Bakke, and Lukas Sekanina. Later, I did some modifications to produce better results to suit the current project needs. Below is a partial algorithm 1 exhibiting the selected thresholding values.

Algorithm 1 HSV Thresholding

```
1: loop: image rows i
2: loop: image cols j
3: if (huePixel > 230 && huePixel <= 255) ||(huePixel >= 0 && huePixel <=
   11) is true then
4:
5:     if (satPixel > 45 && satPixel <= 255) is true then
6:
7:         if (valPixel > 45 && satPixel <= 255) is true then
8:             hsvBin(i,j) = 255
9:         else
10:            hsvBin(i,j) = 0
11:    else
12:        hsvBin(i,j) = 0
13: else
14:    hsvBin(i,j) = 0
15: goto loop
16: goto loop
```

Figure 17 below shows a source image (a) with the thresholded image (b) generated from using the above algorithm. Here the speed limit sign was segmented successfully using the HSV thresholding algorithm.

(a) (b)

Figure 17: (a) Source image and (b) HSV Thresholding output

4.2.3.2 RGB thresholding function

RGB thresholding function acts similar to the HSV function, specific R,G,B values are used as the threshold to rule out all colours except red. Again the selection of these values was based on a "trial-on-error" bases. Every two primary colors when

mixed together they produce a secondary colour which is in the midway between the two primary colours. Accordingly, a combination of values from the three primary colours was selected to achieve an optimum red colour extraction results. The following algorithm 2 illustrates my threshold selection. The values selected for the algorithm was based on a trial and error testing of various test images.

Algorithm 2 RGB Thresholding

1: *loop*: image rows i
2: *loop*: image cols j
3: **if** $(redPixel > 50)$ && $(redPixel - greenPixel > 13)$ && $(redPixel - bluePixel > 13)$ is true **then**
4: rgbBin(i,j) = 255
5: **else**
6: rgbBin(i,j) = 0
7: **goto** *loop*
8: **goto** *loop*

Figure 18 below shows a source image (a) with the thresholded image (b) generated from using the above algorithm. Here the speed limit sign was segmented successfully using the RGB thresholding algorithm discussed above.

(a) (b)

Figure 18: (a) Source image and (b) RGB Thresholding output

4.2.3.3 Grayscale Subtraction Function

The Grayscale subtraction function, subtracts the intensities of the previously converted Grayscale image from the red channel of the source image. This will result in a one channel 8-bits image containing only the red objects. The following algorithm 3 illustrates the idea.

Algorithm 3 Grayscale Thresholding

```
loop: image rows i
loop: image cols j pixel = redPixel − grayPixel
if (pixel < 0) is true then
    pixel = 0
else
    graySub(i, j) = pixel
goto loop
goto loop
```

The following figure 19 illustrates the results of the segmentation by Grayscale intensity subtraction process.

Figure 19: (a) Red channel image, (b) Grayscale image, (c) Segmented output

4.2.3.4 Otsu thresholding function

In Otsu function, the subtracted Grayscale image from the "graysub" function step is passed to the "otsu" function to produce a binary image by performing Otsu threshold on the image. This choice of thresholding method was selected since the Otsu method works very well on Grayscale images as well as it produces very good thresholding results due to it's dynamic thresholding nature.

Otsu thresholding relies on the concept of clustering the image into two classes the foreground and background, and calculates an optimum threshold value that separates the two classes.The calculation method used in this thesis to calculate the separation between the classes is based on computing the maximum inter-class variance via a histogram. The following formula 4.3 illustrates the inter-class variance:

$$\sigma_b^2(t) = \sigma^2 - \sigma_w^2(t)\sigma_b^2(t) = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2 \tag{4.3}$$

Where σ_b is the inter-class variance of the background, σ is the inter-class variance of the foreground and σ_ω is weighted sum of the two class's variance. ω_0, ω_1 are the probabilities of the classes that are separated by the threshold t indicated as weight.

Finally, $\mu 0$ and $\mu 1$ are the mean classes.

The class probability ω_0 and ω_1 are calculated according the following formulas 4.4 and 4.5:

$$\omega_0(t) = \sum_{i=0}^{t-1} p(i) \tag{4.4}$$

$$\omega_0(t) = \sum_{i=t}^{L-1} p(i) \tag{4.5}$$

Where L is the number of bins in the histogram.

The mean class μ_0 and μ_1 are calculated as follows from equations 4.6 and 4.7

$$\mu_0(t) = \sum_{i=0}^{t-1} i\frac{p(i)}{\omega_0} \tag{4.6}$$

$$\mu_0(t) = \sum_{i=t}^{L-1} i\frac{p(i)}{\omega_0} \tag{4.7}$$

With these formulas the optimum threshold value can be calculated based on the maximum value obtained from the inter-class variance computations. The following algorithm 4 shows the technique used to implement the otsu threshold.

Algorithm 4 Otsu Threshold Calculation Method

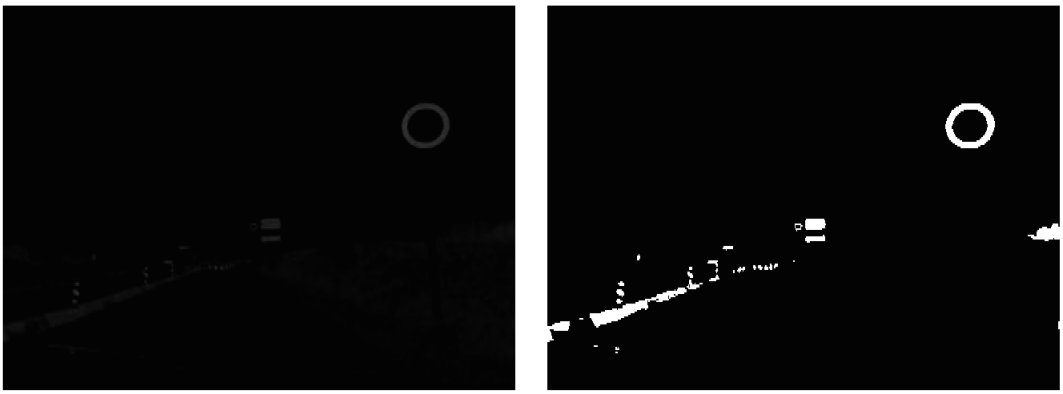
1: Compute histogram
2: Initialize $\omega_i(0)$ and $\mu_i(0)$
3: Iterate through all possible threshold values t and calculate:
4: Compute weights $\omega_0(t), \omega_1(t)$
5: Compute means $\mu_0(t), \mu 1(t)$
6: Calculate maximum $\sigma_b^2(t)$
7: Threshold = maximum $\sigma_b^2(t)$

After the Otsu threshold is calculated, the threshold value is applied to the Grayscale image produced from the segmentation by subtraction "graysub" function to create the binary image. The following algorithm 5 show cases the operation.

Algorithm 5 Grayscale Subtraction Thresholding

```
1: loop: image rows i
2: loop: image cols j
3: if (graysubPixel >= otsuThreshold) is true then
4:     otsuBin(i,j) = 255
5: else
6:     otsuBin(i,j) = 0
7: goto loop
8: goto loop
```

The result of the Otsu thresholding function can be seen in figure 20 (b) below as a binary image.



(a) **(b)**

Figure 20: (a) Segmentation by subtraction Grayscale image, (b) Otsu thresholding binary image

4.2.3.5 Median Filter Function

The median filter function perform noise reduction operations on the output binary images resulted from the thresholding stages. Each of the three outputs is passed sequentially to the "medianfiler" function and produces noise reduced binary images. This function is called inside "hsvThresh", "rgbThresh", and "graysub" functions after performing the thresholding operations.

For the selection of the median kernel, several kernel sizes was experimented on in order to check the optimum noise reduction results produced. And as a result, a 5x5 median kernel was selected for this implementation suitable to be applied images of size 680x480. The 5x5 kernel is placed on the image at the position (0,0) and slides one pixel to the right at each iteration until the kernel slides through the whole image. At each kernel position the median value of the pixels located within the kernel is calculated and the result is written in a new OpenCV image container

corresponding to the original image location.

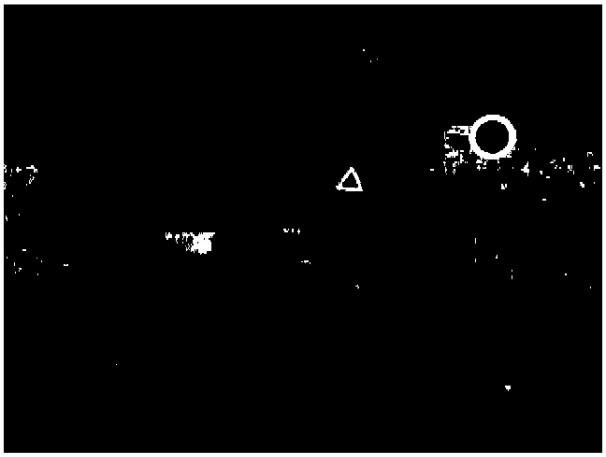
Since the input media is a binary image, there is no need to compute the median based on the conventional calculation methods. It is easier to simply count the number of pixels that have the value of "1" in the binary image corresponding to the kernel window, and if the count is greater than or equal half the kernel size then the median value is set to "1" otherwise, the media is set to "0". The following algorithm 6 illustrates the calculation sequence utilized in the project.

Algorithm 6 Median Filter

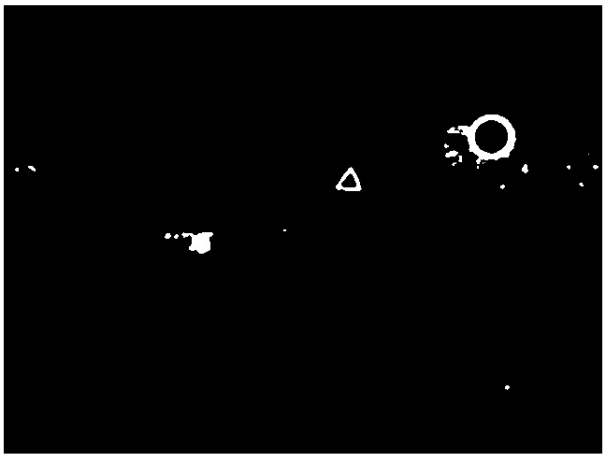
```
1: boundary = kernelSize − 2
2: for i ∈ {i = 1, ..., ImRows − 1} do
3:     for j ∈ {j = 1, ..., ImCols − 1} do
4:         counter = 0
5:         for k ∈ {k = −1, ..., boundary} do
6:             for l ∈ {l = −1, ..., boundary} do
7:                 kernel(1 + k, 1 + l) = binaryIm(i + k, j + l)
8:                 if kernel(1 + k, 1 + l) == 255 is true then
9:                     counter++
10:        if counter >= (kernelSize ∗ kernelSize)/2 is true then
11:            medianDst(i,j) = 255
12:        else
13:            medianDst(i,j) = 0
```

The following figure 21 shows the result of implementing the above algorithm. It can be clearly seen that the median filter algorithm performed well, as the number of artifacts or blobs in the image was reduced by 28% after performing the noise reduction operation.

(a)



(b)



(c)

Figure 21: (a) Original image, (b) Binary image, (c) Binary image after applying median filter

4.2.3.6 Morphological Operators Functions

Erosion and dilation operators was selected in this thesis project to eliminate any possible stranded artifacts or noise that might be still present after applying the median filter. Native OpenCV "erode" and "dilate" methods was called and executed sequentially on the binary image. The binary image was generated from the median filtering stage in the "hsvThresh", "rgbThresh", and "graysub" functions. The structuring element (kernel) size for the erosion and dilation was set to 5x5, and the shape was set to $MORPH\_CROSS$. Figure 22 illustrates the results of the erosion and dilation operation.

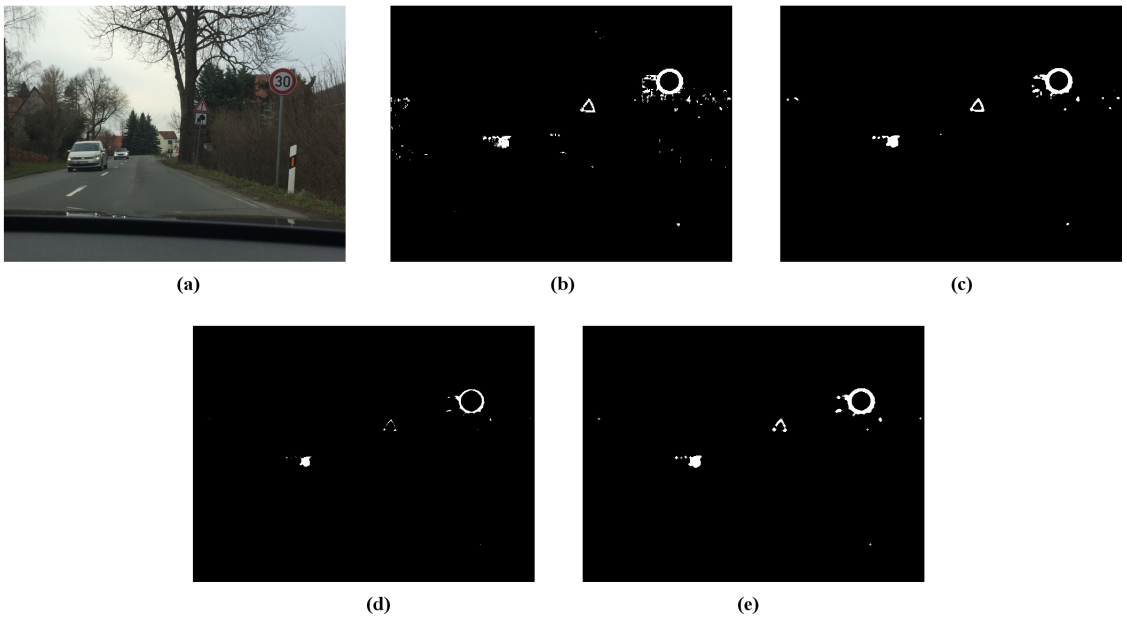


Figure 22: (a) Original image, (b) Binary image, (c) Image after noise reduction, (d) Image after Erosion, (e) Image after Dilation

As illustrated above, figure 22 (d) shows the binary image after erosion operator is executed. Here all the objects in the image was thinned down. Figure 22 (e) shows the eroded image after performing dilation. In this case the objects regained their previous dimensions. The only difference in this setting is, small artifacts and noise were further removed during the erosion operations leading to a cleaner less noisy result in the dilation stage. Dilation is merely used to regain the original dimensions of the speed limit sign.

4.2.4 Merge Function

The merge module merge all three binary images generated from previous stages into one binary image. The merging of the images creates strong robust result, an image that misses a section of the speed limit sign, can be easily compensated from the sections of the other two images. Also this method further eliminates any inferior artifacts that still linger in the images. The merge process is performed by first applying a bitwise operator AND on the HSV and Grayscale subtracted images. An bitwise OR is then applied on the result and the RGB image. Finally, additional bitwise AND is applied on the final result and HSV image. The results of the operation is shown in figure 23 below.

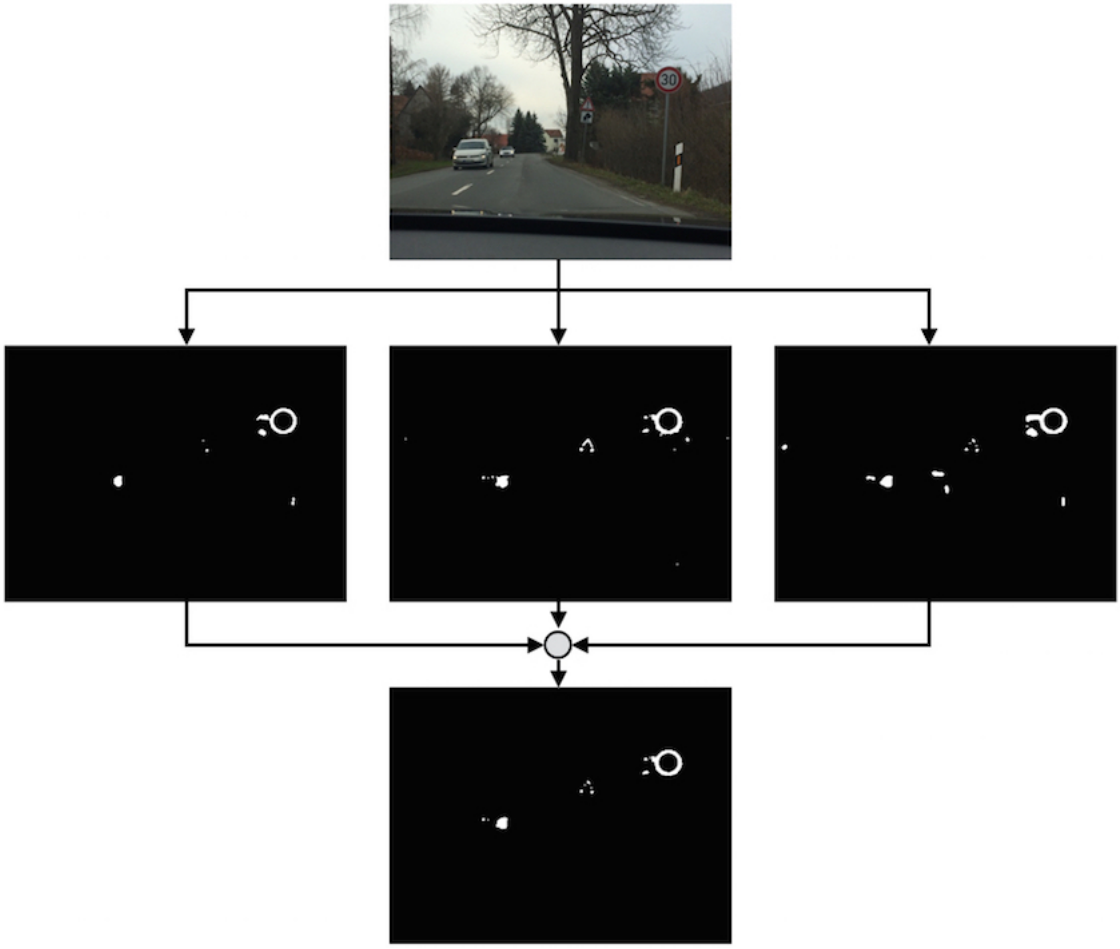


Figure 23: (a) Original image, (b) Binary image, (c) Image after noise reduction, (d) Image after Erosion, (e) Image after Dilation

4.2.5 Downsize Function

The "sCale" function, scales down the binary image from the merge module into 0.5 of it's original size. This approach was utilized in order to reduce the time needed to execute the template matching algorithm in later module. Downsizing will reduce the computational complexity during the similarity matching process, consequently the templates sizes will be reduced as a result. More details about template matching will be discussed next section.

For this work I have selected the bilinear sampling algorithm also known as bilinear interpolation to perform the downscaling (down-sampling). The advantage of this approach it is easy and produce realistic results. However, one disadvantage is that it can produce aliasing artifacts if the algorithm is used to down-sample the image more than a half it's size. To overcome this disadvantage, a pyramid-based approach for downsizing is applied to the image. In this case the algorithm is applied three times to reduce the image with a scale factor of 0.5 from 640x480 then to 320x240 then to 160x120 then to 80x60.

In a binary image the bilinear interpolation considers taking the weighted average of every 2x2 pixel neighborhood (4 pixels) at a time and produces an output to a new image container placing the new averaged pixel result in place of the 2x2 pixel neighborhood. Algorithm 7 shows the implemented method.

Algorithm 7 Bilinear Interpolation Downsizing

1: fx= scaleFactor
2: fy= scaleFactor
3: Resize empty output container "resizedImg" to: (round($fx * binSrcRows$), round($fy * binSrcCols$))
4: **for** $i \in \{i = 0, \ldots, resizedImgRows - 1\}$ **do**
5: **for** $j \in \{j = 0, \ldots, resizedImgCols - 1\}$ **do**
6: pixel = round(($binSrc(2 * i, 2 * j) + binSrc(2 * i, 2 * j + 1)+$
7: $binSrc(2 * i + 1, 2 * j) + binSrc(2 * i + 1, 2 * j + 1))/4$)
8: resizedImg(i,j) = pixel

4.2.6 Template Matching Function

My approach to template matching is to apply simple spatial convolution algorithm to find the similarity between the input image and any of the selected templates. Convolution was explained previously in section 3.1.1.3, basically computing the sum of dot products between the source image and the templates as demonstrated in formula 3.1. The idea is, for each of the available template, convolution is computed and local results are stored in matrices (1 matrix for each each template) as well as the local x and y coordinates for each result point. The following algorithm 8 shows an Next, a local maximum operation is performed to find the maximum

local result value in each result matrix. Finally, a global maximum operation is executed to find the global maximum result value among all result matrices. The A possible similarity matching result is produced by computing the global maximum point value together with their pixel location.

The "templateMatching" function receives the output binary image from the "sCale" function which is passed from the program entry point function "main". At the same time 5 templates are loaded from the PC and prepared for the matching process. Since there are more than one template, the "templateMatching" function creates a list of vectors of CV::MAT type and push each template into the list. Also the function creates additional 3 arrays "List_Local_Results", "List_Local_x", and "List_Local_y" for the local matching results as well as the x and y-coordinates. There are two functions called inside "templateMatching", the "matchingAlg" and "localMax" functions. Where the "matchingAlg" perform the similarity matching computations and "localMax" computes the maximum value of the results obtained from each template matching operation. In this case, the function will compute 5 local maximum results and store them with their coordinates in the "List_Local_Results", "List_Local_x", and "List_Local_y" arrays respectively. The following algorithm 8 shows an excerpt of the executed matching algorithm.

Algorithm 8 Similarity Matching Algorithm

```
1: rowIndex=0
2: colIndex=0;
3: for i ∈ {i = 0, ..., binImRows} do
4:     for j ∈ {j = 0, ..., binImCols} do
5:         out = 0
6:         for k ∈ {k = 0, ..., templCols − 1} do
7:             for l ∈ {l = 0, ..., templRows − 1} do
8:                 rowIndex = i + l
9:                 colIndex = j + k
10:                out+ = binIm(rowIndex, colIndex) ∗ templ(l, k)
11:        result(i,j) = out
```

Algorithm 9 below illustrates how the local maximum result is computed, operated from the "localMax" function.

Algorithm 9 Computing Local Maximum Matching Result

```
1: rows=0
2: cols=0;
3: maxresult= 0
4: for i ∈ {i = 0, ..., resultRows − 1} do
5:     for j ∈ {j = 0, ..., resultCols − 1} do
6:         if maxresult < result(i, j) is true then
7:             maxresult = result(i,j);
8:             rows=i
9:             cols=j
10:    maxLocalresult = maxresult
11:    matchLocalX = cols
12:    matchLocalY = rows
```

Finally, the x and y coordinates generated with the highest matching result is plotted and displayed to the screen using OpenCV's local "rectangle" function. The coordinates are passed to the function together with the dimensions of the matched template (stored previously during the computations of the matching algorithm) to plot a white square on the detected speed limit sign as shown in the below figure 24.

Figure 24: A speed limit sign detection of a 640x480 image

4.3 Hardware Environment

4.3.1 Target Device

The target device selected for the speed limit sign detection system implementation, is the Xilinx Virtex-6 XC6VLX240T FPGA board. Virtex-6 family devices like the one displayed in the figure 25 below have the advantage of producing high-performance clocking capabilities, as well as delivering advance power management

features while maintaining low-power consumption and at lower costs [22]. The Virtex-6 XC6VLX240T FPGA board features:

- Logic cells: 241,152
- CLB: 37,680 slices with maximum 3,650 k-bits distributed RAMs
- DSP48E1: 768 slices
- Block RAM: 416 BRAMs each 36 k-bits, in total maximum of 14,976 k-bits
- I/O: maximum 720
- Transceivers: Gigabit GTX of 24 and GTH of 0

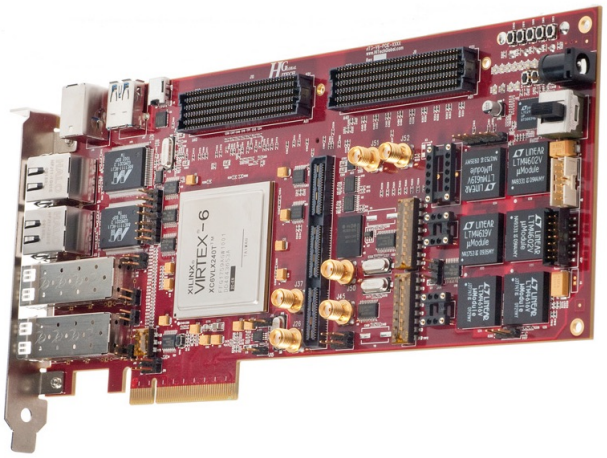


Figure 25: HTG-600 Xilinx Virtex-6 XC6VLX240T development board [23]

4.3.2 Development Environment and verification Tools

The hardware implementation was developed in full version of Xilinx ISE®Design Suite version 14.3. The Design Suite features all basic technologies, tools, and design flows to produce high optimal hardware designs. These includes PlanAhead™ offers RTL to bit stream design flow, ISE Simulator (ISim), Xilinx Platform Studio (XPS), RTL Schemetic viewer, XST Synthesis, CORE Generator for IP core generation, Timing Analyzer, XPower Analyzer for power analysis and consumption, SmartXplorer for multuple implementation flow, and ChipScope™ for FPGA device debugging.

4.4 Hardware Implementation

In this section the hardware implementation of the traffic sign detection will be discussed in details. The system is fully parallelized and pipelined to achieve high-performance, speed, high throughput, and minimum latency. The following sections

I will discuss in depth the design and execution of the individual hardware accelerated components.

4.4.1 Hardware Architecture

As illustrated previously in figure 13, the hardware architecture executes the three preprocessing components in parallel at the same time. As soon as outputs are produced, the segmentation module starts it's computations by providing inputs to the Thresholding, Grayscale Subtraction, and Otsu thresholding components. Once the binary signals are generated, the median filter starts execution followed by the morphological operators. The similarity matching algorithms in the template matching module will run in parallel to produce the required matching results.

The following figure 26 shows the top view schematic of the fully integrated traffic sign detection system design. A 640 x 480 RGB image represented in hexadecimal is passed to the detection system via the preprocessing, segmentation, merge, and template matching modules. After processing the system outputs the matching results, x-coordinates and y-coordinates of the detected image.

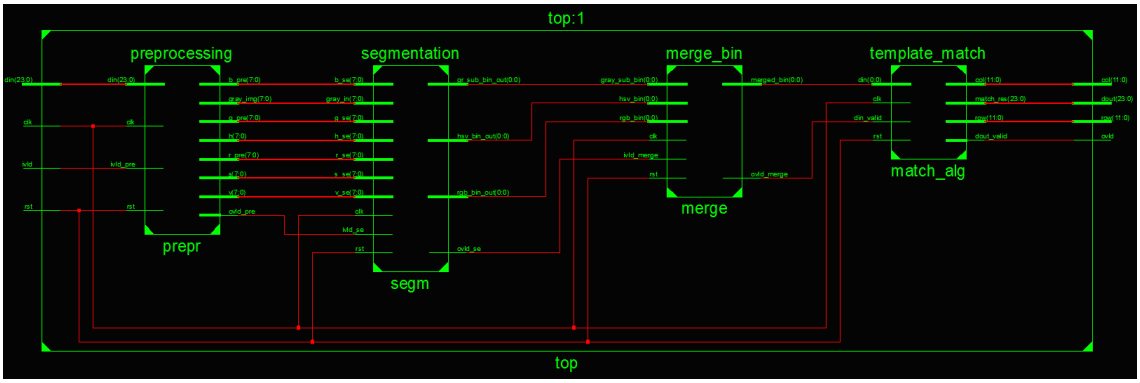


Figure 26: Traffic sign detection system top view RTL schematic

If there are no dependencies between the different components, or requirements to execute the components in series; it is highly recommended to design the hardware accelerated system in parallel. Parallelized hardware architectures decreases overall system latencies and execution time. As a result, the designed hardware structure delivers a concurrent system with high performance and speed. The following figure 27 shows an abstract design-flow of the proposed parallel hardware for the traffic sign detection system. The synchronization buffers are used to synchronize signals in certain modules to generate all the outputs on the same clock cycle .

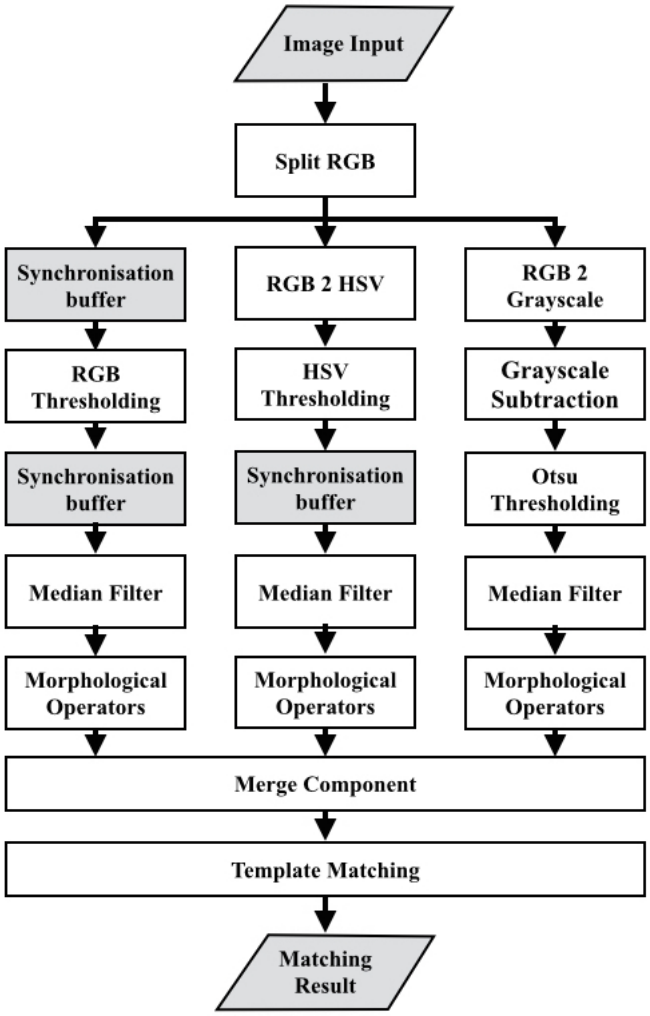


Figure 27: Hardware design flow of the proposed parallel system

4.4.2 Preprocessing

As discussed earlier the preprocessing module performs all necessary preprocessing operations needed to ensure efficient segmentation and detection execution in later stages. This module is responsible for RGB channels split, RGB to HSV conversion, and RGB to Grayscale conversion. The following image 28 shows an RTL design wrapper for the module where it consist of split RGB and RGB conversion components. The input for the component is 24-bits rgb image data (din) streamed in a Raster manner. There are clock (clk) and reset (rst) input signals for synchronization, in addition to a valid data input (ivld_pre) for signal verification. The outputs of the module are red (r), green (g), blue (b), blue (h), saturation (s), value (v), and Grayscale (gray_img) 8-bits signals, in addition to an output valid signal (ovld_pre).

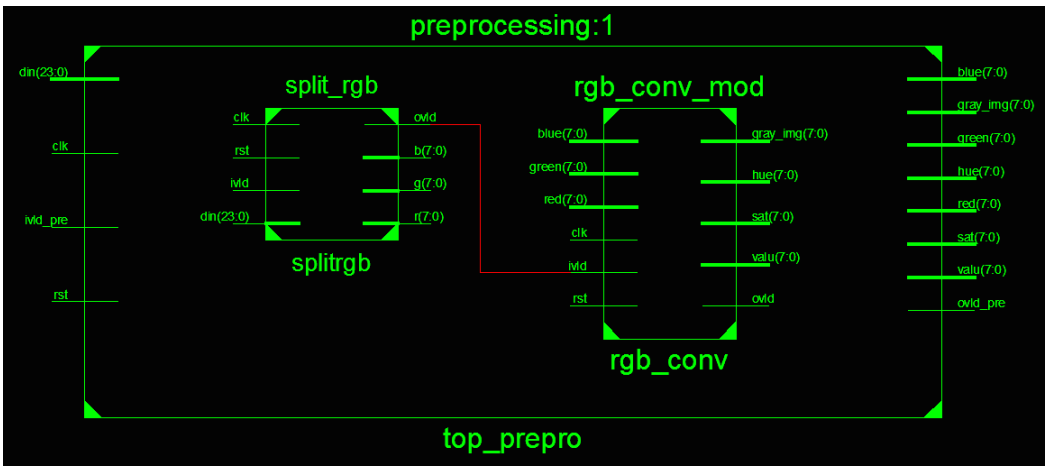


Figure 28: Preprocessing module RTL schematic

The Split RGB component divides the incoming 24-bits RGB data into three signals red, green, and blue each are 8-bits long. The colour signal extraction process follows the BGR channel arrangement, as the image hex data was extracted from OpenCV library which supports the BGR sequence.

The RGB conversion sub module consists of two components, the RGB to HSV converter and RGB to Grayscale converter. The calculations executed for the RGB to HSV component are based on the formulas discussed in Chapter 5 section 5.2.2. In this module, native DSP Intellectual Property (IP) cores to perform basic arithmetic operations which are used for the saturation and hue computations. The IP cores uses the DSP48E1 slices available for the Virtex-6 FPGA devices, the IP cores are generated using Xilinx LogiCORE Generator and imported into the design for further utilization. The advantage of using DSP48E1 slices IP cores over the traditional methods, is that they are designed to achieve highest optimization level either according to area or speed [22]. In the case of this thesis design optimization for speed was selecetd although it will increase the latency of the design. The DSP48E1-based based IPs relies on fully pipelining the computational stages, hence increasing overall performance of the system. In addition, in the designed module itself is fully parallelized and pipelined through introducing intermediate registers between the different computational stages as seen in figure 29. This modules utilizes 4 multipliers, 4 dividers, and 4 subtracters DSP48E1-based IP cores.

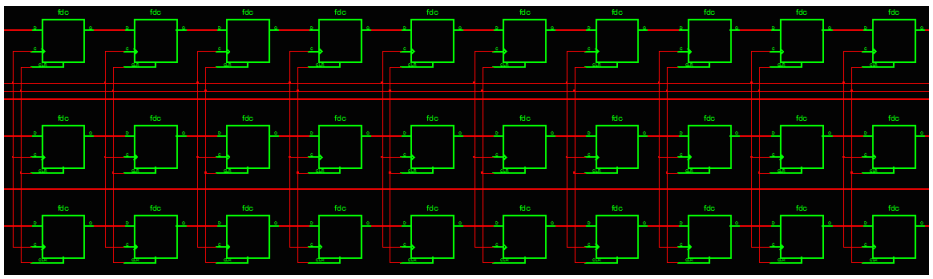


Figure 29: Section of pipelined stages in RGB to HSV module

The RGB to Grayscale component takes the three 8-bits signals red, green, and blue produced from the Split RGB component and converts them into one 8-bits Grayscale signal. The RGB to Grayscale conversion calculation is based on the following formula 4.8

$$Grayscale_{image} = 0,3 * Red_{pixel} + 0,59 * Green_{pixel} + 0,11 * Blue_{pixel} \tag{4.8}$$

From the hardware perspective to avoid working with floating points the fractions from the above formula was approximated to: $0,25 * Red_{pixel} + 0,5 * Green_{pixel} + 0,125 * Blue_{pixel}$. The new formula was then implemented into to the hardware by shift right 2-bits the red signal, shift right 1-bit the green channel, and shift right 3-bits the blue signal. Pipelined stages through registers was introduced between the shifts operation of the red and green channel to synchronism the output of each operation. The results are then added together to produce the final output representing the Grayscale signal.

The results of the preprocessing module (red, blue, green, hue, saturation, value, and Grayscale signals) must to be synchronized at the same time for the component to export the output successfully to the next stage. To achieve that, the system adds pipeline stages specifically for the red, green, blue channels.

4.4.3 Segmentation

The segmentation module is designed to perform colour-based segmentation via RGB and HSV thresholding. The module also performs Grayscale intensity subtraction from the red channel followed by otsu thresholding. After thresholding the segmentation module runs noise reduction algorithms by applying binary median filter. Followed by morphological operations for further reduction of noise and small artifact. The figure 30 displayed below shows a parallel implementation of the segmentation module where both the thresholding (RGB and HSV) module and Grayscale subtraction module runs in parallel producing the outputs as the same time. Moreover, the noise reduction and morphological operations execute in parallel for each of the thresholding outputs as indicated in the figure.

The module's takes the results produced from the preprocessing modules. These are 8-bits red (r), green (g), blue (b), hue (h), saturation (s), value (v), and the Grayscale (gray_in) channels. After processing, the module outputs 3 binary 1-bit signals from the hsv thresholding (hsv_bin_out), rgb thresholding (rgb_bin_out), and the Grayscale subtraction binary (gr_sub_bin_out).

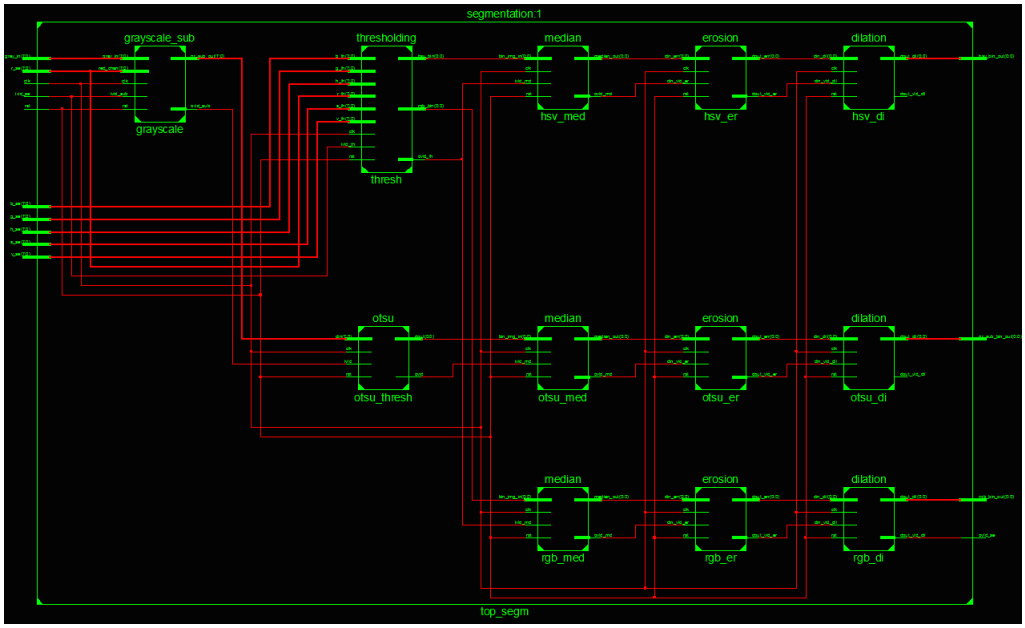


Figure 30: Segmentation module RTL schematic

4.4.3.1 Thresholding Module

The thresholding module receives the 8-bits red, green, blue, hue, saturation, and value signals from the preprocessing module, perform thresholding and output two 1-bit binary signals. The module consists of two VHDL processes designated for the RGB and HSV operations. Also the module uses 2 IP cores subtracters for the RGB thresholding process.

Referencing the values from algorithm 2 in chapter 4 section 4.2.3.2, a 3-bits lookup table (LUT) was designed to perform the RGB thresholding operation in hardware. The LUT approach provides fast and reliable method to produce results in a timely manner as well as reduces latencies during system design. with conditional statements, registers, and multiplexing the whole thresholding process took 3 clk cycles to process.

To perform the thresholding operation a 3-bits register signal is used to set the output to high (1) only when any of the red, green, or blue values are above the threshold values of 50 for red, 13 for $red - green$, and 13 for $red - blue$. This

operation is done every clock cycle for the duration of the whole input data. The following image 31 illustrates the RGB thresholding using the LUT.

Input			o/p
1	1	1	1
Otherwise			0

Figure 31: RGB thresholding LUT

The HSV thresholding process is similar to the RGB thresholding process. The difference is the LUT table used for the HSV is 8-bits long and the thresholding formulas are executed from algorithm 1 in chapter 5 section 5.2.3.1. An 8-bits register signal is used as a flag for the LUT which sets the output to high (1) only when the values of hue is between (230-255) and (0-11), saturation between (45-255), value between (45-255). Each of these 8 value ranges designates 1-bit in the 8-bits register signal flag in which an if statement validates the current input each clock cycle. The following figure 32 shows the LUT for the implemented HSV thresholding.

Input								o/p
1	1	1	1	0	1	1	1	1
1	1	1	1	1	0	1	1	1
1	1	1	1	0	0	1	1	1
1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	0	1	1
1	1	1	1	1	1	0	0	1
Otherwise								0

Figure 32: HSV thresholding LUT

A one pipeline stage (1 clock cycle) was introduced into the two processes to delay the output results of both the RGB and HSV thresholding. This delay is intentional, as later on, the thresholding module will produce synchronized outputs with the outputs of the Otsu thresholding module.

4.4.3.2 Otsu Thresholding Module

In the Grayscale subtraction module the operation is performed using a subtracter IP to subtract the Grayscale signal from the red signal. The output is then for-

warded to the Otsu thresholding module to perform thresholding and to generate a 1-bit binary signal output.

As explained previously in an earlier chapter, Otsu threshold calculation method depends on dynamic threshold computations. In this thesis I utilized a previously implemented module for Otsu threshold value calculation[1]. Therefore, by using the module I generated the threshold values of the test images separately and used it directly in my design to perform thresholding on the input Grayscale signal. I preferred obtaining the threshold values directly than to incorporate the module into my design for a reason. The test image must be evaluated twice by the system, once to calculate the Otsu threshold value and the second time to apply the threshold value to the image. This will introduce very large latency to the design as well as it will consume large number of resources since histogram calculation is a complex procedure.

4.4.3.3 Median Filter Module

This module receives the binary signals produced from the thresholding operations and perform noise reduction using the median filter. The module is designed based on a 5x5 kernel sliding window which will take in the input binary signal and apply the median filtering operation to the values inside the window[1]. Every clock cycle the data input is shifted one bit to the left, theoretically this will make the window slides over the image one pixel to the right. As discussed previously, the module sums the number of (1s) inside the kernel window every clock cycle and if the sum is greater than or equal the size of half of the kernel dimension (ie.13) then median (median_out) is set to (1) otherwise is set to (0). The following figure 33 demonstrates the logic operation of the binary median filter.

[1]The component was developed by Stephan Blokzyl, Department of Computer Engineering at Chemnitz University of Technology (TUC)

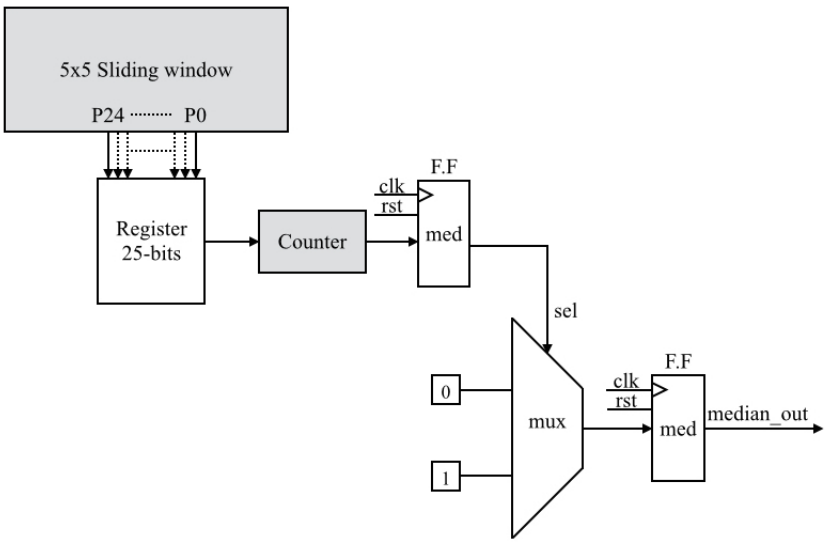


Figure 33: Binary median filter logic operation

4.4.3.4 Erosion and Dilation Modules

The erosion module takes in the binary output of the median filter and perform the erosion operation on the data accordingly. Like with the median filter, the module uses a 5x5 sliding window[1], also known as structuring element for the processing of the input data. I have selected a cross shape kernel window to perform the operation. The following figure 34 illustrates the pixels arrangements of the structuring element. When the window slides over the pixels of the image, pixels that have the value of 1 and corresponding to the pixels in the sliding window cross formation will be flagged as "fit" which is what is needed for the erosion. To achieve that in hardware a 9-bits LUT is created using a 9-bits register signal as the flag, Whenever there is a "fit" a 1-bit from the register flag will be set to 1, otherwise will be set to 0. Next, if the 9-bits flag register has all 1s then the erosion output signal will be set to 1 otherwise 0. The choice of the 9-bits LUT results from the 9 pixels positions which have 1s resembling the cross shape.

0	0	1	0	0
0	0	1	0	0
1	1	1	1	1
0	0	1	0	0
0	0	1	0	0

Figure 34: A cross shape kernel window illustration

Since dilation is the opposite operation of erosion, the hardware implementation of the dilation module is very similar to that of the erosion's. The only difference is, instead of checking if the register flag contain 1s the algorithm checks if it has 0s, accordingly set the dilation output signal to 1 otherwise set it to 0.

4.4.4 Merge Image Module

This module merge all three binary images extracted from the dilation operations of the colour-based and Otsu-based thresholding to form one robust binary image. The resulted image shows the region of interest (ROI), the speed limit sign that will be later forwarded to the template matching module for detection. The module performs bitwise operation on the three binary signals. First, apply bitwise AND between the segmentation by subtraction binary and the HSV binary Then, apply bitwise OR between the result and the RGB binary. Finally apply bitwise AND between the result and HSV binary. This combination of using bitwise OR and ANDs produces images with less noise and/or inferior artifacts while maintaining the strong robust features of the ROI. The figure 35 below illustrates the RTL schematics of the discussed module.

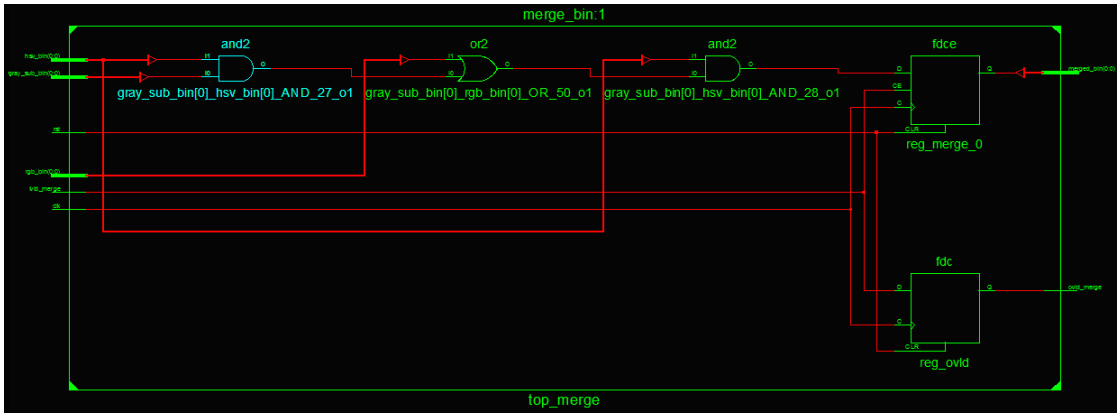


Figure 35: RTL schematic of the merge module

4.4.5 Template Matching Module

The detection method of the speed limit sign starts with the template matching module. This module performs similarity matching operations on the segmented binary image produced from previous stages. As discussed previously template matching simply try to match an image template or a set of templates to a source image, aiming to find similarities between the template and the source image. This module is composed of 5 similarity matching components corresponding to the 5 chosen templates. Each component executes individually and in parallel a separate similarity matching algorithm. The matching algorithm is based computing spatial convolution between the source binary image and one of the templates. The tem-

plates dimensions are 4x4, 5x5[1], 6x6, 7x7, and 9x9. In addition, each component computes the highest value of the local matching results computed by the similarity matching algorithm.

The template matching module receives the merged binary image signal input from the merge component, and outputs the highest matching results (match_res), as well as the x-coordinates and y-coordinates of the best possible match. The highest matching result is generated from calculating the global maximum of the 5 highest similarity matching results, producing the possible template matching result together with the coordinate location of the template. Figure 36 below shows the RTL wrapper for th template matching module.

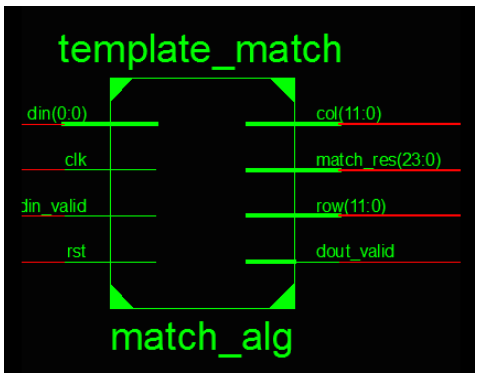


Figure 36: RTL schematic wrapper of the template matching module

Each of the similarity matching component perform the computations similarly, the only difference is the dimension of the template kernel window used. I have extended and regenerated four binary sliding windows of dimensions 4x4, 6x6, 7x7, and 9x9 from the already existed 5x5 sliding window [1]. The 5 sliding windows are convoluted over the binary source image to compute the matching results, location coordinates, and the highest local result in their respective components. The sliding windows are designed based on using a certain number of registers aligned next to each other. Also certain number of FIFOs were utilized per window for data buffering between each window line. This arrangement based on each template dimension will create a NxN sliding window as illustrated in the following image 37.

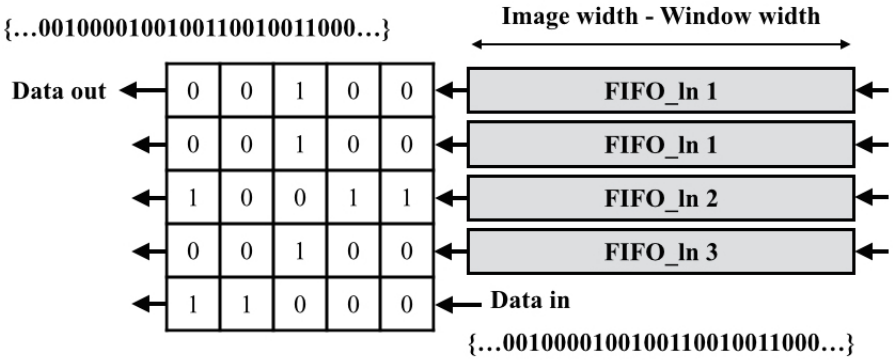


Figure 37: 5x5 sliding window illustration

Each matching component utilizes certain number of Block ROMs (BROM) generated from the IP Core generator. The ROMs are used to store the template binary data. In my design I choose to use one ROM for each line in a template (each bit in the line represents a pixel), for example for the 4x4 template I used four 4-bits wide ROMs, for the 5x5 template I used 5-bits wide ROMs, and so on. The reason for this selection, I need to read the complete template binary data for each clock cycle. This will allow the operation to be synchronized with the sliding window outputs for the convolution computations later on.

Spatial convolution can be easily computed on binary images with less complexity. For each similarity matching process of each template, template data stored in the ROMs (e.g. 4x4 template) are read first. Knowing that each 1-bit in a ROM line represents 1 image pixel, a bitwise AND is applied between each ROM line and it's corresponding image pixels in the sliding window for every line. And the results of each line are stored in temporary register signals. In the second step, for each temporary register signals each bit is added together. And every addition operation is assigned to new temporary register signals. Finally in the third step, the matching results is computed when all the lines are summed together. Subsequently, the algorithm calculates the highest matching results for each component as well as the local x and y-coordinates of the results.

Two counters were used to calculate the location coordinates of the matched results. One counter is used to count the number of columns (incremented each clock cycle), and one counter for counting the number rows (incremented every image width). The counters are triggered by flags that keep track of each local computed result of the convolution process in a pipelined manner.

5 Evaluation and Results

A software prototype of the traffic sign detection system was first developed and tested to verify the correctness and complexity of the algorithms. The implementation was executed on a MacBook pro running Windows 10 with 2,7 GHz Intel Core i7 and 4 GB 1333 MHz DDR3 of RAM. The design was developed in Eclipse Mars using in C++ and OpenCV library version 2.4.13. In addition, the system was visualized using OpenCV "rectangle" function to plot the location of the match and displayed to screen. After the software implementation was completed and verified, the hardware hardware accelerated implementation stared in VHDL. The hardware design and implementation is based on real-time data streaming of the image from the internal PC hard drive. Only five binary templates were stored on board of the FPGA on BROMs. However, due to time limitations in the thesis time frame, the hardware accelerated system was only simulated in Xilinx ISE Simulator (ISim) and synthesized but not deployed and tested on the FPGA device.

In this chapter I will discuss the evaluation and results of the traffic sign detection system for both the hardware and software implementation. It should be noted that it is very important to perform validation and verification on the system to ensure that the system delivers the required results according to the requirements and specifications of the design. In the next section details regarding the test data and templates will be discussed.

5.1 Test Data

Since this is an image processing design, the test data set is composed of 20 images of different speed limit signs taken in German streets and highways. The images were captured in high resolution with dimensions of 3264 x 2448 and are later downsized to 640x480 for the testing activities. The selected images are captured in different conditions such as over exposed lighting, under exposed lightning, normal lightning, snowy conditions, sunny conditions, faded sign, as well as an image with an occluded object.

Originally six binary template images were chosen, showing the outer circular form of a typical German speed limit sign. The six template images are of different sizes ranging from 32x32, 38x38, 46x46, 52x52, 62x62, 78x78 pixels as shown in below figure 38. These templates were downsized by half three times since the larger dimensions will increase implementation and computational complexity, as

well as larger dimensions will require lot of logic and memory resources from the FPGA hardware. During the template size reduction, it was noticed that two of the templates became almost of the same size, and as a result I removed one of them making the final template count to be 5 templates of dimensions 4x4, 5x5, 6x6, 7x7, and 9x9.

Figure 38: Speed limit sign templates of different dimensions [24]

In the hardware implementation, the templates stored in the IP BROMs are saved in a binary form inside coefficient files COE. The COE files are generated and configured with binary data of the bit matrix by the Xilinx CORE Generator™. The below figure 39 illustrates the 5 binary templates stored in the BROM. Each line of each matrix were stored in a separate BROM as discussed previously in chapter 5 section 5.3.5.

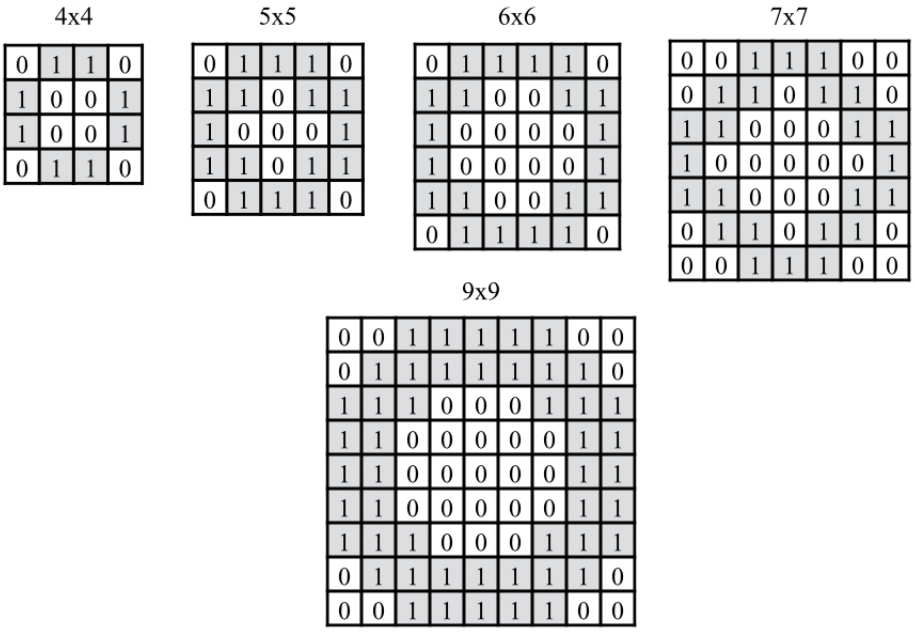


Figure 39: Six binary templates of different dimensions of a speed limit sign

The RGB image pixel values were converted into hexadecimal[1] notation imported to Xilinx ISE®design suite in the form of a package in a VHD file format. After

[1]The conversion process was performed using an RGB to hexadecimal conversion program I have previously developed in c++

successful import the data is read into the traffic sign detection system input port signal for processing.

5.2 Software Implementation Test Results

As discussed previously the speed limit sign detection system is based on the template matching technique which is computed using a typical spatial convolution computations. The below figures 40 and 41 illustrates the detection results from executing the implemented system. The test set consists of images taken from different weather and lightning conditions, as well as images with faded signs, occlusion and false positives. Fourteen out of the twenty images showed positive results for sign detection with a success rate of 70%. Figure 40 (b) shows two connecting signs a warning and a speed limit signs, the system was able to detect only the speed limit sign despite the presence of the warning sign in the segmented output image. Figure 40 (c) shows image taken on a bright day, the speed limit sign is partially overexposed in some areas. However, the detection algorithm was able to find the sign in the correct position. Figure 41 (d) the image was taken in a snowy condition with partial occlusion of the sign with snow. The system was able to detect the sign despite the partial segmentation of the ROI. Although the matching algorithm was able to detect the sign in figure 41 (d) however, the algorithm matched the sign with the incorrect template size (can be observed from the large unfitted square around the sign).

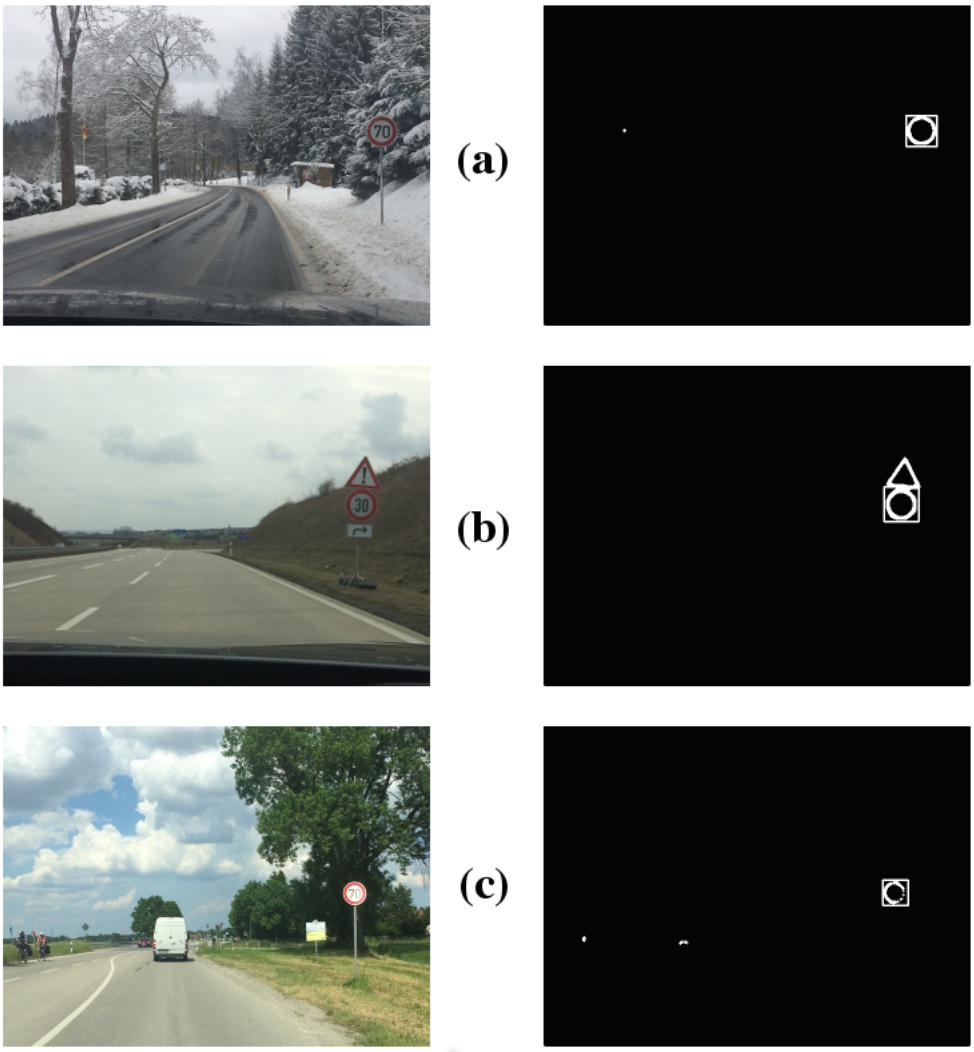


Figure 40: Successful detection of speed limit signs

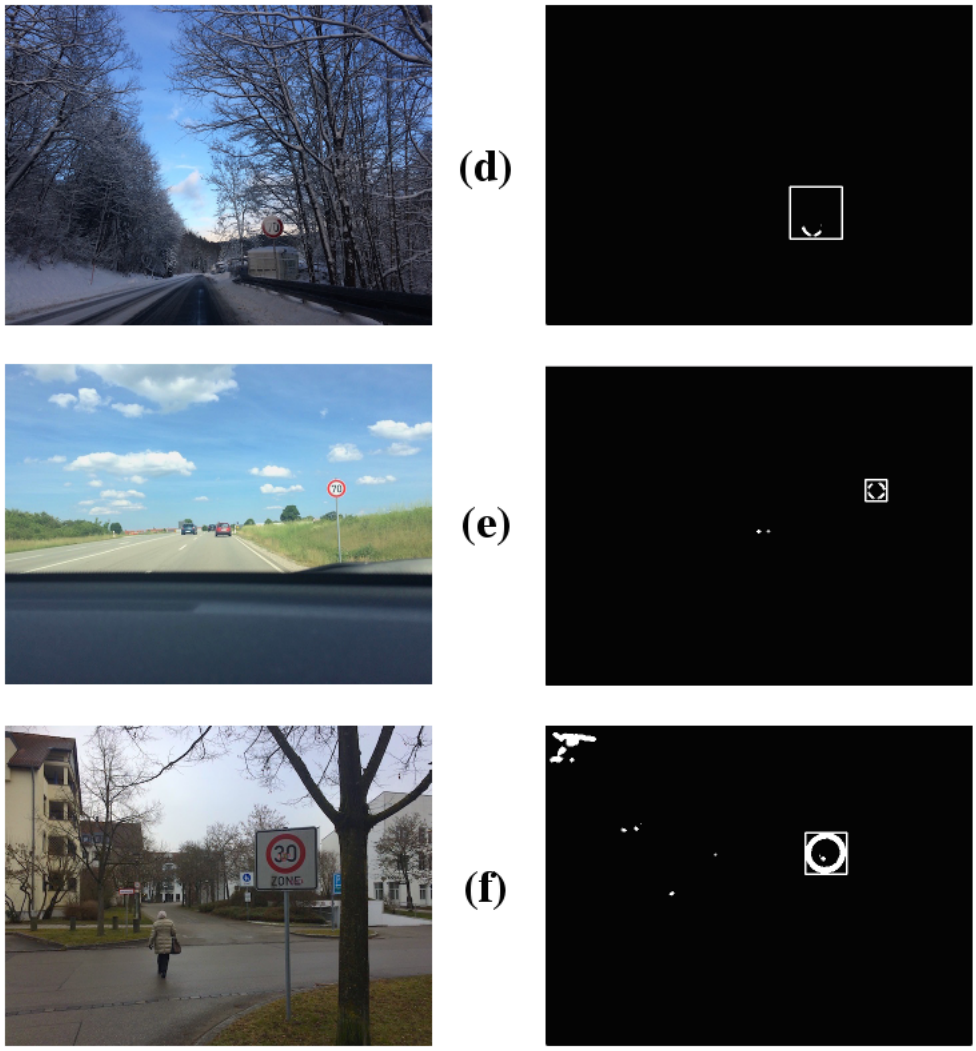


Figure 41: Successful detection of speed limit signs

In 3 out of twenty images (15%), the matching algorithm detected false positives in the images. False positives corresponds to objects which are not of interest to the requirements of the system but otherwise detected. For example large objects like cars, roof tops, other red signs, etc... pass the segmentation phase and goes through the detection algorithm. On the other hand small sized inferior objects or artifacts were easily removed during the noise reduction and morphological operations phases or not detected by the algorithm at all. The below figure 42 demonstrates the 4 images where the sign detection system detected false positives instead of the true positive the speed limit sign.

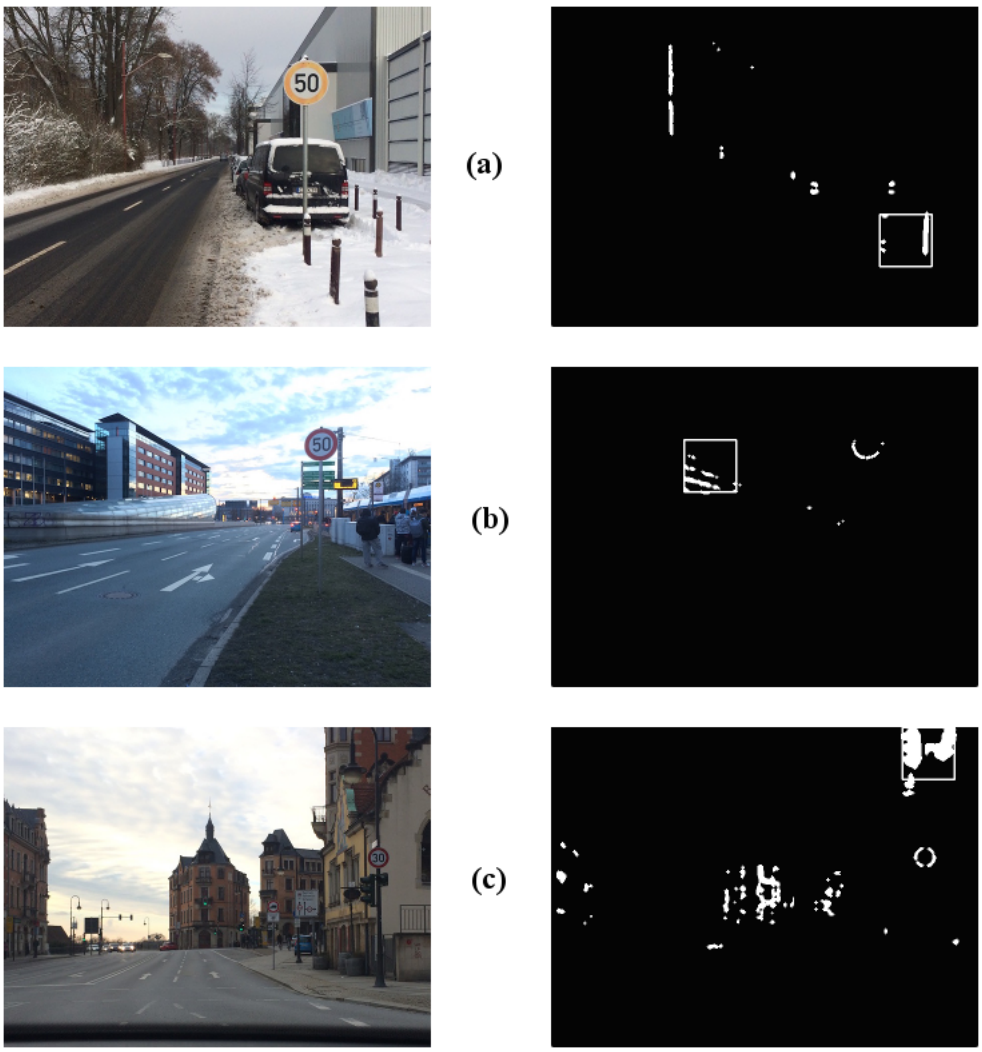


Figure 42: False positive detection

Figure 42 (a) shows an image with a faded speed limit sign due to weather conditions, it did not pass through the segmentation phase however, the template matching algorithm system detected a small pole instead. Figure 42 (b-c) illustrated different images with different lightning conditions that have the speed limit signs segmented, with inferior objects detected instead. Finally, in three out of twenty images (15%) the traffic signs were not detected at all by the implemented system.

The overall software system computational time executed approximately in 51 seconds for 640x480 images, and approximately in 0.81 seconds for the downscaled version 80x60 (executed by the downsize component).

5.3 Hardware Implementation Test Results

As noted previously the hardware implementation was only simulated in Xilinx ISE Simulator (ISim). The hardware accelerated design consists of 20 components excluding the top level module. A unit test was performed on each component separately, followed by system integration tests of mid-level components. And finally, tests for the complete integrated system on the top level module. Fixed point computations were used in the thresholding module calculations. This allow to reduce computational complexities of the operations leading to increase in system performance.

The results from testing the hardware components yielded in a 87% success rate. Two components produced inaccurate results due to an error occurred during the implementation process. The first component is the RGB to Grayscale conversion, and the second component is the template matching component. As a consequence, the overall system produced imprecise detection results.

Latency is the delay or time frame in clock cycles that takes a system to compute an operation. Table 5.1 below shows the individual latencies for each component operating at 100MHz frequency for images of 640x480 pixels.

Component	Latency (clk cycles)
Top Design Module Level	12964
Preprocessing	27
Split RGB	1
RGB Conversion	26
RGB to HSV	26
RGB to Grayscale	26
Segmentation	7705
Thresholding	4
Grayscale Subtraction	3
Otsu Thresholding	1
Median Filter	2567
Erosion	2567
Dilation	2567
Merge Binaries	1
Template Matching	5231
Match 4x4	1964
Match 5x5	2617
Match 6x6	3270
Match 7x7	3923
Match 9x9	5229

Table 5.1: Module based latencies

5.3.1 Performance Analysis and Synthesis Results

The hardware performance of the proposed system is evaluated in terms of resource utilization and operating frequency. This evaluation depends mainly on the nature of the architecture of the selected hardware device, in this case the Virtex-6 xc6vlx240t-2-ff1759 FPGA. The design was synthesized and simulation for the post-route was executed. The Place and Route (PAR) report indicated that the maximum operating frequency for the design is 72,2 MHz (14.027 ns).

In their paper, K. Mikolajczyk and C. Schmid [**?**] measured the processing time of a given system, by dividing the number of clock cycles required to complete a task by the operational frequency of the hardware. The following formula **??** estimates the processing time based on:

5.3.1.1 Resource Utilization

Resource utilization is a very important subject in hardware design optimization, the less resources used the better the performance and speed of the system. Three important resource utilization parameters to consider, are the number of utilized Flip-Flops, LUT and memory in the form of Block RAM/FIFO. The following table 5.2 shows the resource utilization summary of the traffic sign detection system generated by the synthesis report for the Viretx-6 xc6vlx240t-2-ff1759 FPGA with a speed grade of -2. The proposed system only utilized 4% of the available LUTs and 11% of the available memory on the. Memories such as Block ROMs and FIFOs was extensively used in the design to create sliding windows of different dimensions, as well as to store the five binary templates used in template matching.

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	8,499	301,440	2%
Number of Slice LUTs	7,451	150,720	4%
Number of LUT Flip-Flop pairs used	10,545		
Number of RAMB18E1/FIFO18E1s	93	832	11%
Number used as Memory	49	58,400	1%
Number of bonded IOBs	76	720	10%
Number of DSP48E1s	11	768	11%

Table 5.2: Synthesis report resource utilization summary

The numbers and percentages displayed in the table above indicates that the proposed traffic sign detection design can be implemented successfully on low-cost, low-budget FPGA devices without the need to worry about running out of resources.

6 Conclusion and Future Work

For the past two decades the main focus in the Automotive industry is to design and develop fast and efficient advance driving assistance systems in particular traffic sign recognition systems (TSR) to help and guide drivers through their daily commute. And for that reason the aim of this master thesis is to design and develop an FPGA-based hardware accelerated solution for traffic sign detection focusing on German speed limit signs. The system should be fast and robust enough to detect the speed limit signs in despite weather and lightning conditions, and obstacles.

The proposed system was designed and implemented in both software and hardware environments. The software implementation was merely used as a prototype for the hardware design and to test and verify the correctness and effectiveness of the selected algorithms. The proposed hardware accelerated system, presents a fully parallelized and pipelined solution which provides high speed, flexibility, and high performance results as compared to designs developed in the traditional software platforms. The hardware implementation is designed to perform at a clock frequency of 100 MHz operating on a Virtex-6 XC6VLX240T FPGA board using Xilinx ISE Design Suite and ISE Simulator (ISim) with VHDL as the hardware description language.

The traffic sign detection system passes through three main development stages: preprocessing, segmentation, and detection. The main detection technique selected for this work is template matching. Template matching is a set of feature detection algorithms which searches for similarities in images based on a smaller image called template. The algorithms can use more than one template during the execution of the algorithm on a given image. The evaluation of the proposed system showed promising results, the system performed very well in software with a successful detection rate of 75% out of the 20 test images used to test the system. This includes speed limit signs located in various weather and lightning conditions, as well as signs occluded with snow. The overall detection processing time in software took around 51 seconds give or take for 640x480 images. Also for the downscaled version 80x60 (executed by the downsize component), the processing time took around 0.81 seconds. In the hardware implementation, unit tests was executed and simulated heavily on each component (16 VHDL modules) by supplying the modules with individual test hexadecimal and binary data, delivering output with a success rate of 87%. As for the system integration testing, all components were integrated successfully through Xilinx ISim simulation producing results after 14.486ns.

There are timing issues with software implementations, due to the nature of the general purpose sequential instruction-based environments. Software-based architectures do not have real-time operating systems, and as such they do not expected to have a dedicated processing time from the CPU just for only one application. Therefore, latencies and delays can be expected specially when the CPU shifts to other processes. On the other hand, FPGA based devices do not have the problem of shifting to other processes. Due to the nature of FPGA device architectures, they are a dedicated reconfigurable logic hardware that run designs in parallel delivering real-time and high performance results. In addition, latencies in FPGA devices due to propagation delays between flip-flops and internal logic merely ranges in microseconds. For that reason hardware based designs produces superior performance over traditional software based architectures. For the proposed system the overall simulation system design latency was realized in 12964 clock cycles.

6.1 Limitations

In this work limitations and restrictions was introduced during the design and implementation process of the traffic sign detection system.

Image sizes used was restricted to be minimum of 640x680 pixels due to the current templates size limitations used in the hardware implementation. Templates were of sizes 4x4, 5x5, 6x6, 7x7 and 9x9 and larger images with large signs won't be detected. The reason for this template dimension selection is due to the high complexity of the hardware implementation of the template matching algorithm, when template sizes gets larger, the algorithm will require more resources and computational time.

Other major limitation, due to the thesis time limitation and the complexity of the system design and the implementation on both software and hardware (16 VHDL components), the traffic sign hardware accelerated detection system was only implemented to run in Xilinx ISim simulation. However, the synthesis tool was able to run the design successfully producing a full synthesis report. In addition the downsizing algorithm was not implemented in hardware again due to the timing constraints of the thesis.

Other challenges in the design is the sensitivity of the colour thresholding in the segmentation stage, despite HSV known to be invariant to lighting conditions, there were some difficulties detecting the signs in very bright and dark images, same results was observed for the RGB thresholding as well. Also the system was not able to eliminate false positives in images leading to false positive detection in some of the images. Finally the system was not able to detect more than one sign at the same time in an image, due to the thresholding of the maximum global matching results during the template matching computations.

6.2 Recommendations and Future Work

This thesis work opens future work for some of the present implementation issues limiting the traffic sign detection system. I will be providing some recommendations that can improve the performance and efficiency of the proposed algorithms and techniques to produce more robust and efficient system. Speed limit detection: For detecting more than one sign in an image, it is recommended to provide during template matching a range of highest similarity matching results, by eliminating just selecting one maximum matching result. Also, additional algorithms should be added to the system to eliminate false positive during segmentation process, such as eliminate artifacts based on shape, area, width, and length sizes. RGB and HSV thresholding: Further adjustments to the thresholding algorithm values to improve the segmentation phase and to produce more significant and robust results that are light and weather invariant. Finally, high resolution images and video streams: Expend the hardware implementation to accommodate live streaming of high resolution real-time image and video streams captured from camera sensors and run full live demonstration on the FPGA device.

Bibliography

[1] *Virtex-6 FPGA Configurable Logic Block*, "Xilinx, 2012.

[2] J. Torresen, J. W. Bakke, and L. Sekanina, *Recognizing Speed Limit Sign Numbers by Evolvable Hardware*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 682–691.

[3] H. Gomez-Moreno, S. Maldonado-Bascon, P. Gil-Jimenez, and S. Lafuente-Arroyo, "Goal evaluation of segmentation algorithms for traffic sign recognition," *IEEE Transactions on Intelligent Transportation Systems*, vol. 11, no. 4, pp. 917–930, Dec 2010.

[4] K. Brkić, "An overview of traffic sign detection methods," 2010.

[5] R. Hmida, A. Abdelali, and A. Mtibaa, "Hardware implementation and validation of a traffic road sign detection and identification system," *Journal of Real-Time Image Processing*, pp. 1–18, 2016.

[6] S. B. Wali, M. A. Hannah, S. Abdullah, A. Hussain, and S. A. Samad, "Shape matching and color segmentation based traffic sign detection system," *Przeglad Elektrotechniczny*, vol. R. 91, no. 1, pp. 36–40, 2015.

[7] S. Blokzyl, M. Vodel, and W. Hardt, "A hardware-accelerated real-time image processing concept for high-resolution eo sensors," in *Proceedings of the 61. Deutscher Luft- und Raumfahrtkongress*, Berlin, Germany, September 2012.

[8] D. KOC. (2015) Deutschland verkehrszeichen. [Online; accessed May 15, 2017]. [Online]. Available: http://deutschlandverkehrszeichen.blogspot.de/2015/08/verkehrszeichen-deutschland-download.html

[9] Wikimedia, "Rgb color solid cube," 2008, [Online; accessed May 15, 2017]. [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:RGB_color_solid_cube.png&oldid=214095084

[10] A. de la Escalera, J. M. Armingol, J. M. Pastor, and F. J. Rodriguez, "Visual sign information extraction and identification by deformable models for intelligent vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 5, no. 2, pp. 57–68, Jun 2004.

[11] Wikimedia, "Hsl hsv cylinder color solid comparison," 2008, [Online; accessed May 15, 2017]. [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:HSL_HSV_cylinder_color_solid_comparison.png&oldid=193805514

[12] C. G. Keller, C. Sprunk, C. Bahlmann, J. Giebel, and G. Baratoff, "Real-time recognition of u.s. speed signs," in *2008 IEEE Intelligent Vehicles Symposium*, Jun 2008, pp. 518–523.

[13] C. Souani, H. Faiedh, and K. Besbes, "Efficient algorithm for automatic road sign recognition and its hardware implementation," *Journal of Real-Time Image Processing*, vol. 9, no. 1, pp. 79–93, 2014.

[14] P. Elfert, T. Tiemerding, C. Diederichs, and S. Fatikow, "Advanced methods for high-speed template matching targeting fpgas," in *2014 International Symposium on Optomechatronic Technologies*, Nov 2014, pp. 33–37.

[15] K. Briechle and U. D. Hanebeck, "Template matching using fast normalized cross correlation," in *Proc. SPIE 4387, Optical Pattern Recognition XII*, vol. 4387, March 2001, pp. 95–102.

[16] J. P. Lewis, "Fast normalized cross-correlation," in *Canadian Image Processing and Pattern Recognition Society, Quebec City*, may 1995, pp. 120–123.

[17] X. Zhou, Y. Ito, and K. Nakano, "An efficient implementation of the one-dimensional hough transform algorithm for circle detection on the fpga," in *2014 Second International Symposium on Computing and Networking*, Dec 2014, pp. 447–452.

[18] J. Illingworth and J. Kittler, "The adaptive hough transform," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-9, no. 5, pp. 690–698, Septemebr 1987.

[19] G. Loy and A. Zelinsky, "Fast radial symmetry for detecting points of interest," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 8, pp. 959–973, August 2003.

[20] M. A. V. Toro, "Fast radial symmetry detection for traic sign recognition," Master's thesis, University of Applied Sciences Bonn-Rhein-Sieg, Sankt Augustin, Germany, 2015.

[21] I. Xilinx. (2012) Virtex-6 fpga configurable logic block user guide. ug364.pdf. [Online; accessed May 15, 2017]. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug364.pdf

[22] K. Hashimoto, Y. Ito, and K. Nakano, "Template matching using dsp slices on the fpga," in *2013 First International Symposium on Computing and Networking*, Dec 2013, pp. 338–344.

[23] HitechGlobal, "Htg-600: Xilinx virtex 6 pci express gen 2 / sfp / usb 3.0 development board," 2017, [Online; accessed May 15, 2017]. [Online]. Available: http://www.hitechglobal.com/Boards/Virtex6_FPCIExpress_Board.htm

[24] J. Torresen, J. W. Bakke, and L. Sekanina, "Efficient recognition of speed limit signs," in *Proceedings. The 7th International IEEE Conference on Intelligent Transportation Systems (IEEE Cat. No.04TH8749)*, Oct 2004, pp. 652–656.

[25] N. Aguirre-Dobernack, H. Guzmn-Miranda, and M. A. Aguirre, "Implementation of a machine vision system for real-time traffic sign recognition on fpga," in *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, Nov 2013, pp. 2285–2290.

[26] "Hardware/software co-design of a traffic sign recognition system using zynq fpgas," *Electronics*, vol. 4, no. 4, pp. 1062–1089, 2015.

[27] H. Fleyeh, C. Sprunk, S. O. Gilani, and M. Dougherty, "Road sign detection and recognition using fuzzy artmap: A case study swedish speed-limit signs," in *Proceedings of the 10th IASTED International Conference Artificial Intelligence and Soft Computing*, Aug 2006, pp. 242–249.

[28] S. Saboya, "Matlab-based implementation for detection of possible speed limit signs," Master's thesis, Technische Universität Chemnitz, Chemnitz, 1992.

[29] I. Andreadis, "A real-time color space converter for the measurement of appearance," *Pattern Recognition*, vol. 34, no. 6, pp. 1181–1187, 2001.

[30] T. Hamachi, H. Tanabe, and A. Yamawaki, "Development of a generic rgb to hsv hardware," in *The Proceedings of the 1st International Conference on Industrial Applications Engineering 2013*, Fukuoka, Japan, Mar. 2013, pp. 169 – 173.

[31] N. K. Ratha, A. K. Jain, and D. T. Rover, "Convolution on splash 2," in *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, CA, USA, Apr 1995, pp. 204–213.

[32] A. Benedetti, A. Prati, and N. Scarabottolo, "Image convolution on fpgas: the implementation of a multi-fpga fifo structure," in *Proceedings. 24th EUROMICRO Conference (Cat. No.98EX204)*, vol. 1, aug 1998, pp. 123–130.

[33] K. A. ElDahshan, M. I. Youssef, and M. A. Mustafa, "Hardware segmentation on digital microscope images for acute lymphoblastic leukemia diagnosis using xilinx system generator," *(IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 5, no. 9, pp. 33–37, 2004.

[34] S. Sural, G. Qian, and S. Pramanik, "Segmentation and histogram generation using the hsv color space for image retrieval," in *Proceedings. International Conference on Image Processing*, vol. 2, sep 2002, pp. II–589–II–592 vol.2.

[35] J. R. Smith and S.-F. Chang, "Single color extraction and image query," in *Proceedings., International Conference on Image Processing*, vol. 3, Oct 1995, pp. 528–531.

[36] X. Wang, R. Hnsch, L. Ma, and O. Hellwich, "Comparison of different color spaces for image segmentation using graph-cut," in *2014 International Conference on Computer Vision Theory and Applications (VISAPP)*, vol. 1, Jan 2014, pp. 301–308.

[37] "Introduction," in *Reconfigurable Computing*, S. Hauck and A. Dehon, Eds. Burlington: Morgan Kaufmann, 2008, pp. xxv – xxix.

[38] C. W. Yu, J. Lamoureux, S. J. E. Wilton, P. H. W. Leong, and W. Luk, "The coarse-grained / fine-grained logic interface in fpgas with embedded floating-point arithmetic units," in *2008 4th Southern Conference on Programmable Logic*, March 2008, pp. 63–68.

[39] J. Ni, M. K. Singh, and C. Bahlmann, "Fast radial symmetry detection under affine transformations."